MEDICAL EQUIPMENT
FEBRUARY 2025
ISSUE 415

circuitcellar.com

circuit cellar

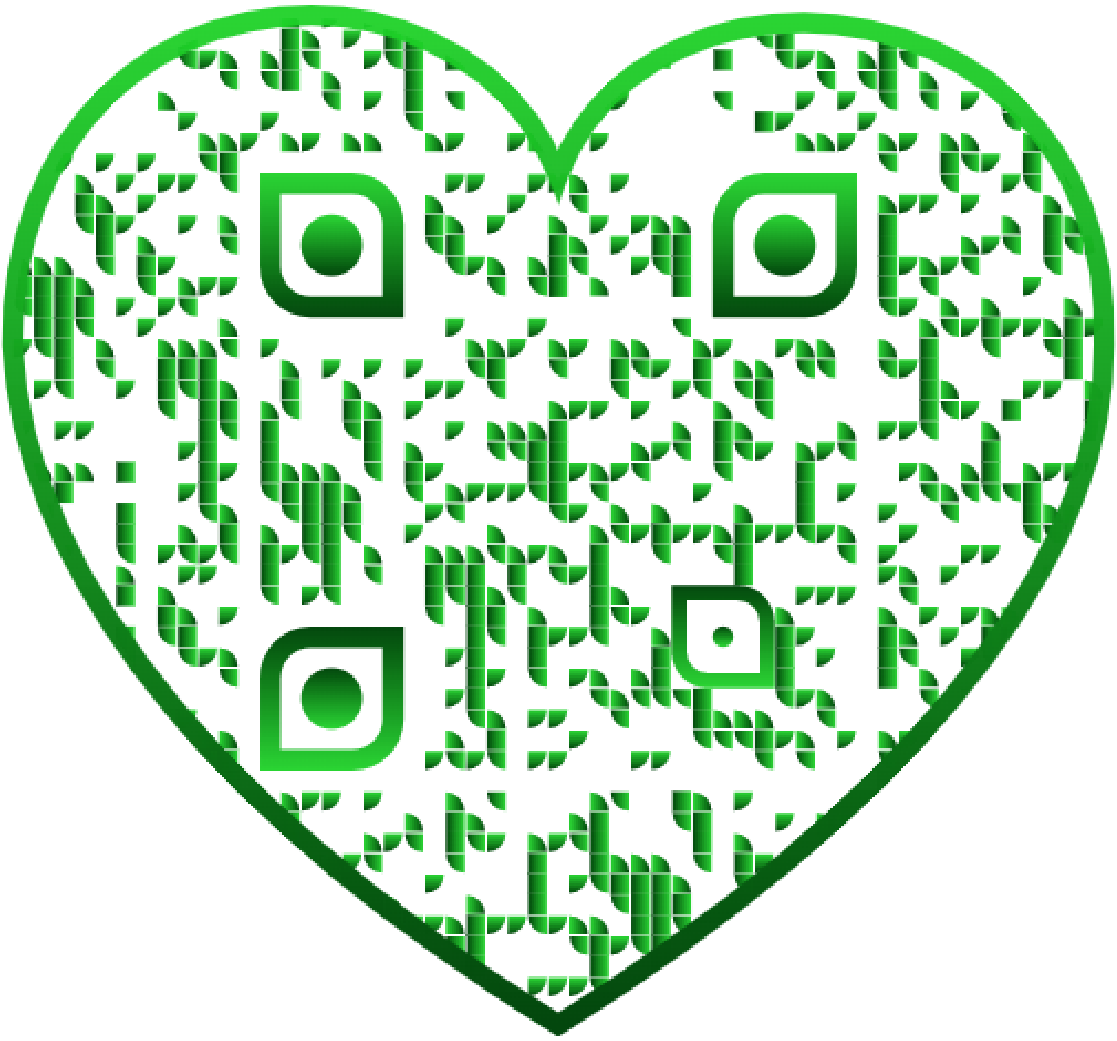
Inspiring the Evolution of Embedded Design

ENGINEERING HEALTHCARE

PACING CONTROL
Mode
Sync
500
THERAPY CONTROL
End Procedure
VOLTAGE
2.0
Prepare System
Confirm
Deliver Energy
CONFIRM
DELIVER
Total Deliveries 3

System-on-Module Technology | Unlock Bluetooth LE Audio | Take on LISP Programming with MCUs | Revive Retro Audio via Raspberry Pi Pico | Basic Pulse Circuits Metastability and Setting | CMake the Most of Software Development | Regulator Efficiency — Minimize Power | Asynchronous Programming in Flutter | The Future of Development Boards

$9.00US $10.00CAN



Imagine what you will
Discover
circuit cellar
VAULT
CC VAULT
circuit cellar
SMART BUILDINGS AND CITIES
circuit cellar
DATA ACQUISITION TECHNOLOGY
circuit cellar
PCB DESIGN & PROTOTYPING
circuit cellar
TINY EMBEDDED BOARDS

400+ ISSUES

Get the complete *Circuit Cellar* issue archive and article code stored on a stylish, durable and portable USB flash drive. You can easily keep your CC Vault archive up to date by purchasing subsequent issues from our webshop or by downloading issues with a *Circuit Cellar* Digital Subscription. Issues appear in searchable PDF format.

cc-webshop.com

circuit cellar
WEARABLE DEVICES

circuit cellar
AI AND ROBOTICS

circuit cellar
DRONE DEVELOPMENT AND USAGE

circuit cellar
MCU DEVELOPMENT TOOLS

Complete archive includes PDFs of all issues in print through date of purchase.

INPUT Voltage

Why New Habits Are Worth the Trouble

We are creatures of habit, whether we like it or not. The tendency to become accustomed to a routine or process seems universal as one of the ties that bind us as humans. However, when things are efficient and easy, we close the door to all other opportunities. That could be a mistake that prevents us from realizing our full potential.

Take a build system, for example. This is a set of software tools that automates the process of converting source code into executable programs, libraries, or other software components. Build systems are a vital part of software development and can help improve productivity, consistency, and reliability.

An article on build systems reminds me of getting into a routine with blinders.

In his first of a bi-monthly series of articles entitled *Modern Embedded Insights*, Jacob Beningo, an expert in embedded system programming, reminds us that most developers start their projects with the MCU's Integrated Development Environment (IDE). Yes, it is quick and efficient, and likely, these tools are very familiar to the people who are building with vendor-provided tooling. However, some efficiencies and benefits can be acquired for those who dare to shed old, accustomed ways. In "CMake the Most ofSoftware Development" Jacob reminds us that adopting a new methodology can modernize any developer's build system and open the door to greater possibilities for scaling, linking libraries, debugging, and more.

Also, regarding this issue, we offer a review of System-on-Module technology in our Datasheet feature. Check out the newest offerings from Aaeon, Axiomtek, BeagleBoard, Nvidia, Toradex, Variscite, SolidRun, and even the latest Raspberry Pi CM5. Processors steal the spotlight of many SoMs, and here, you'll find representatives from Intel, AMD, NXP, Broadcom, and Nvidia. Not only are SoMs key in the growth of IoT technology, but they are particularly crucial in adopting Industry 4.0 standards.

Does anyone remember the Frogger arcade game? Returning author Chris Cantrell not only remembers it, but he brings the distinctive sounds of the game back to life with a classic AY-3-8910 sound chip and a Raspberry PI with a Zilog Z80 emulator in his project, "Reviving Frogger's Retro Audio with a Programmable Sound Generator."

The latest Bluetooth Low Energy Audio specifications are about to revolutionize audio technology like Stereo sound did in the 1950s. Nick Hunn makes his Circuit Cellar author debut by describing the new audio middleware and isochronous channels. The article is Introducing Bluetooth LE Audio, the first part of a two-part series.

In the second installment of a two-part series, Raul Torrico delves into the world of micro Lisp or "µLisp" and shows how to extend its capabilities by adding custom functions to enable the interface with ultrasonic sensors and control servo motors.

Wolfgang Matthes concludes a three-part series of articles on basic pulse circuits to provide digital designers with insight for processing asynchronous signals from sensors and level translators. The great thing about this series is how Wolfgang assembled knowledge for his subject that was not found in textbooks or coursework. He gleaned information from the supporting literature accompanying numerous MCUs, microprocessors, and FPGAs and presented it in a casual yet scholarly fashion.

All of the selections above were carefully curated for you, our devoted readers, so that you can enhance your skills and knowledge for your endeavors. If you have a project you are working on and think it may help others, we'd love to hear about it. It could lead to the next insightful *Circuit Cellar* article, enticing developers to dispense with old habits and adopt new ones.

Tom Murphy
Editor-In-Chief
tmurphy@circuitcellar.com

Issue 415 February 2025 | ISSN 1528-0608

CIRCUIT CELLAR® (ISSN 1528-0608) is published monthly by:

KCK Media Corp.
PO Box 417, Chase City, VA 23924

Periodical rates paid at Chase City, VA, and additional offices. One-year (12 issues) subscription rate US and possessions $50, Canada $65, Foreign/ ROW $75. All subscription orders payable in US funds only via Visa, MasterCard, international postal money order, or check drawn on US bank.

SUBSCRIPTION MANAGEMENT

Online Account Management: circuitcellar.com/account
Renew | Change Address/E-mail | Check Status

CUSTOMER SERVICE

E-mail: customerservice@circuitcellar.com

Phone: 434.533.0246

Mail: Circuit Cellar, PO Box 417, Chase City, VA 23924

Postmaster: Send address changes to
Circuit Cellar, PO Box 417, Chase City, VA 23924

NEW SUBSCRIPTIONS

circuitcellar.com/subscription

ADVERTISING

Contact: Hugh Heinsohn

Phone: 757-525-3677

Fax: 888-980-1303

E-mail: hheinsohn@circuitcellar.com
Advertising rates and terms available on request.

NEW PRODUCTS

E-mail: product-editor@circuitcellar.com

HEAD OFFICE

KCK Media Corp.
PO Box 417
Chase City, VA 23924
Phone: 434-533-0246

COPYRIGHT NOTICE

Entire contents copyright © 2025 by KCK Media Corp. All rights reserved. Circuit Cellar is a registered trademark of KCK Media Corp. Reproduction of this publication in whole or in part without written consent from KCK Media Corp. is prohibited.

DISCLAIMER

KCK Media Corp. makes no warranties and assumes no responsibility or liability of any kind for errors in these programs or schematics or for the consequences of any such errors printed in Circuit Cellar®. Furthermore, because of possible variation in the quality and condition of materials and workmanship of reader-assembled projects, KCK Media Corp. disclaims any responsibility for the safe and proper function of reader-assembled projects based upon or from plans, descriptions, or information published in Circuit Cellar®.

The information provided in Circuit Cellar® by KCK Media Corp. is for educational purposes. KCK Media Corp. makes no claims or warrants that readers have a right to build things based upon these ideas under patent or other relevant intellectual property law in their jurisdiction, or that readers have a right to construct or operate any of the devices described herein under the relevant patent or other intellectual property law of the reader's jurisdiction. The reader assumes any risk of infringement liability for constructing or operating such devices.

© KCK Media Corp. 2025 Printed in the United States

OUR NETWORK

circuit cellar
LinuxGizmos
audioxpress
VOICE COIL
LIS LOUDSPEAKER INDUSTRY SOURCEBOOK

SUPPORTING COMPANIES

Black Magic	9
BusBoard Prototype Systems	47
CCS, Inc.	61
EmbeddedTS	61, C4
Embedded World 2025	25
Microchip	31
Renesas	60
SeaFire Micros, Inc.	C3

NOT A SUPPORTING COMPANY YET?

Contact Hugh Heinsohn
hugh@circuitcellar.com,
Phone: 757-525-3677,
Fax: 888-980-1303
to reserve space in the
next issue of *Circuit Cellar.*

THE TEAM

FOUNDER Steve Ciarcia
PUBLISHER KC Prescott
CONTROLLER Chuck Fellows
EDITOR-IN-CHIEF Tom Murphy
CONTRIBUTING EDITOR Brian Millier
CONTENT MANAGER- EU Jurij Mikeln
SENIOR ASSOCIATE EDITOR Shannon Becker
TECHNICAL COPY EDITOR Carol Bower
PROJECT EDITORS Ken Davidson
David Tweed
MARKETING MANAGER Tori Zienka
ADVERTISING COORDINATOR
Diana Winsor
ADVERTISING SALES REP.
Hugh Heinsohn

COLUMNISTS
Stuart Ball (Start to Finish)
Jacob Beningo (Modern Embedded Insights)
Bob Japenga (Embedded in Thin Slices)
Brian Millier (Picking Up Mixed Signals)
Colin O'Flynn (Embedded Systems Essentials)

FEATURES

4 **Introducing Bluetooth LE Audio**
Part 1: Sound of a Revolution
By Nick Hunn

10 **Learn Lisp Programming Using MCUs**
Part 2: Extending µLisp with Custom Functions
By Raul Alvarez-Torrico

18 **Reviving Frogger's Retro Audio with a Programmable Sound Generator**
Using an AY-3-8910 Chip and a Z80 Emulator on a Raspberry Pi
By Chris Cantrell

26 **Basic Pulse Circuits**
Part 3: Metastability, Setting, or Resetting
By Wolfgang Matthes

32 TECHNOLOGY FEATURE
Engineering Healthcare
Cutting-Edge Technologies to Enhance Patient Care
By Caleb Smith

38 DATASHEET
System-on-Module Technology Key IoT Enabler
High-Performance Computing Systems Packed into Compact Form Factors
By Tom Murphy

COLUMNS

42 Modern Embedded Insights
CMake the Most of Software Development
Modern Build Systems Support Scalability
By Jacob Beningo

48 Start to Finish
Switching Regulator Efficiency
Minimize Power Consumption in Battery-Operated Systems
By Stuart Ball

54 Embedded in Thin Slices
Asynchronous Programming in Flutter
Shallow Dive into Google's Mobile App Framework
By Bob Japenga

62 TECH THE FUTURE
The Future of Development Boards
NXP MCX MCUs Cut Development Time
FRDM Platforms Revolutionize System Innovation
By Mónica Cid

59 : PRODUCT NEWS
60 : TEST YOUR EQ

@editor_cc
@circuitcellar
circuitcellar
CircuitCellar

Introducing Bluetooth LE Audio

Part 1: Sound of a Revolution

By
Nick Hunn

Following the Bluetooth Core 5.2 release, Bluetooth Low Energy Audio introduces capabilities that will revolutionize audio applications through isochronous channels and new audio middleware. Nick writes about a framework supporting bidirectional audio and shared listening in Part 1 of a two-part article.

Most developers working with Bluetooth specifications focus on the release of the core specification. That's the foundation of Bluetooth, defining the radio, the protocols that transport data over the air, and the fundamental security mechanisms. The first release, Version 1.0, was back in 1999, when the Core included everything up to the application layer, including audio. When new applications were developed, these moved out into individual application profiles, which evolved independently over time. Significant new features in the Core resulted in new prime numbers for subsequent releases. Core 2.0 introduced enhanced data rates, potentially trebling the throughput. Core 3.0, now deprecated, allowed alternate radios, such as 802.11, to achieve higher data rates. Core 4.0 introduced the biggest innovation in 2010, adding Bluetooth LE, which introduced a redesigned radio to support much lower power applications.

Ten years later, in 2020, the release of Core 5.0 started the path to enabling audio over Bluetooth LE, with key innovations appearing in the following two releases. 5.1 brought in extended advertising, which would be key to broadcast applications, and 5.2 added isochronous channels, which enable audio transfer over a Bluetooth LE radio [1].

Anyone who was just following the Core releases might well have missed what was going on outside it. In 2013, a new working group set out to rewrite how Bluetooth supports audio. The two most popular audio profiles—the Advanced Audio Distribution Profile (A2DP) and the Hands-Free Profile (HFP)—have become ubiquitous and are used for audio streaming and telephony, respectively. However, they had major limitations. They were both monolithic specifications, which only worked for a limited number of use cases. Neither supported separate earbuds, which we would see proliferate a few years later. Nor did they provide a simple way to move between audio streaming and telephony, let alone add in voice recognition or audio sharing. In addition, the SBC codec, which provided a base level of audio interoperability, sounded somewhat dated. It was not as efficient as current codecs in the required bitrates or the resulting subjective audio quality. Bluetooth needed a new approach to support a new generation of audio applications. That required the new Core features of 5.x, a new codec, and an independent audio middleware that would be flexible enough to support the next twenty years of audio innovation.

GENERIC AUDIO FRAMEWORK

After an initial analysis to see if the classic Bluetooth approach could be extended, it became apparent that future audio needed a clean sheet. So many different audio directions appeared that the only realistic way to support them was to construct a modular framework where developers could configure the audio streams and audio control however they wanted. Fundamental to that was the principle that the audio data and control planes would be separated, allowing developers to combine them however required.

The other change, which required the 5.2 update of the Core, was to provide topologies that would allow either connected audio data transfer between a small number of paired devices or broadcast audio transmission, where a single transmitter could send audio to any device within range. This latter concept derived from the hearing aid industry, where inductive loops called telecoils broadcast audio to any compatible hearing aid within the confines of that loop. That's a technology that has been largely unchanged for 70 years. Bluetooth provided a natural evolution with the advantage that it would work with any receiving device, bringing audio sharing to a much larger audience. As the consumer electronics and smartphone industries began understanding the possibilities, they became enthusiastic advocates. Broadcast audio, which we will cover in Part Two, is seen as the hallmark feature of Bluetooth LE Audio, being promoted under the Auracast brand. The new specifications that comprise Bluetooth LE Audio are shown in **Figure 1.**

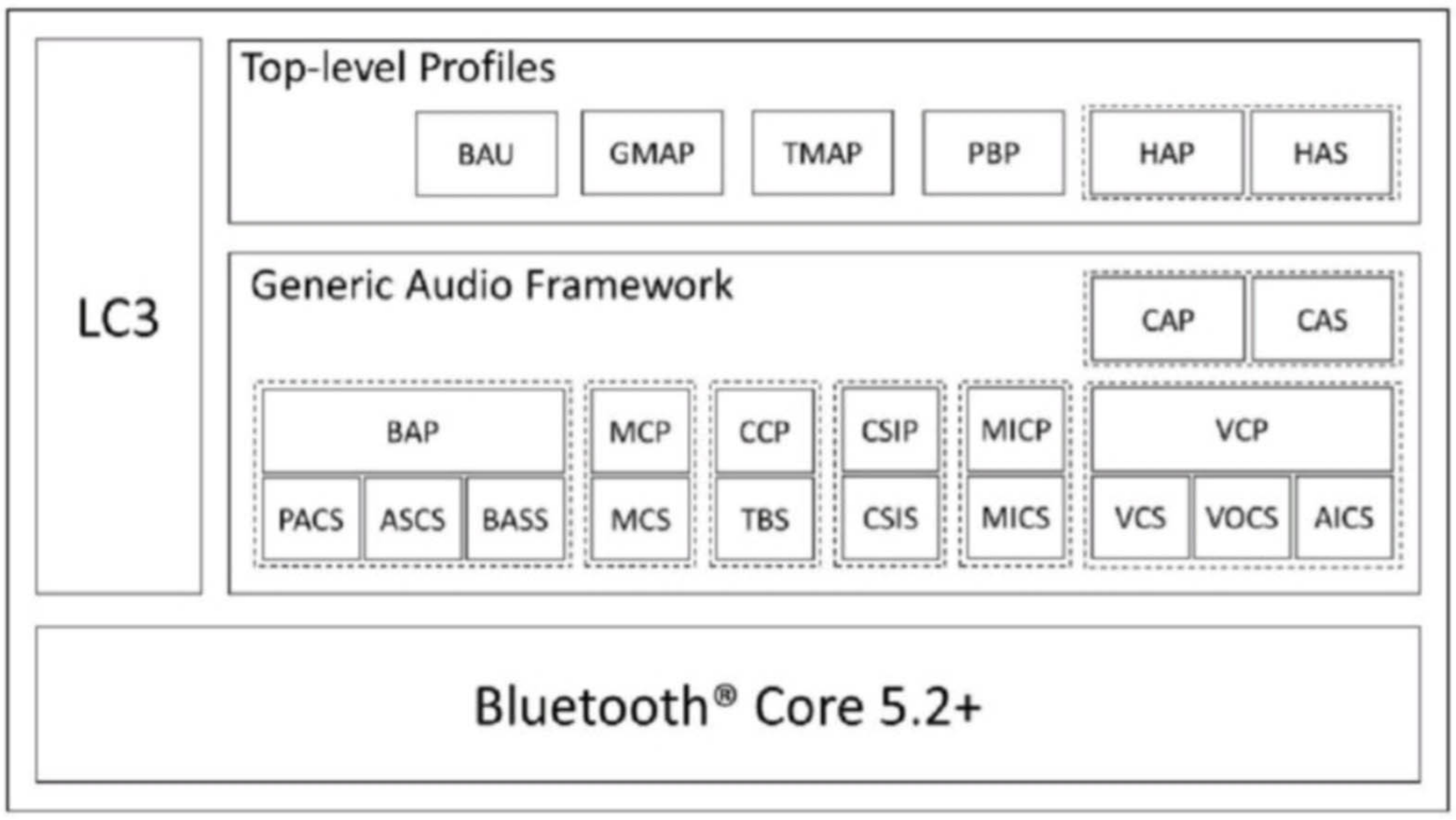


FIGURE 1
The new specifications in Bluetooth LE Audio drive new capabilities in audio applications, which are set to revolutionize the industry.

The new specifications in Bluetooth LE Audio drive new capabilities in audio share that could revolutionize the industry.

At the bottom of the stack, we still have the Core specification, which defines isochronous channels using Bluetooth LE to transport audio data. These can be set up as unidirectional, bidirectional, or broadcast and configured to cope with various codec configurations. They are configured according to the Generic Audio Framework (GAF) specifications, which consists of 18 new specifications covering the isochronous stream configuration, content and telephony control, volume and microphone control, and coordination and transitions. At the top are a growing number of what are termed Top Level Profiles, which define how these GAF specifications are used for specific use cases such as Telephony and Media streaming and new applications, including hearing aids, gaming, and broadcast. The purpose of the GAF is to provide a toolkit that allows developers to address a wide range of audio applications that are far greater than possible with the previous profiles. As we're running on Bluetooth LE, all of these specifications, both in GAF and at higher levels, consist of a Profile and one or more Services, using standard GATT procedures to write to Service Characteristics in a standard Client-Server manner. That becomes tenuous with broadcast, where there is no connection, but the terminology is retained for convenience.

The GAF has also been architected to overcome one of the major limitations of classic Bluetooth audio: the multi-profile issue. In the past, audio streaming with A2DP was designed as a separate specification from telephony, which used HFP. Transitioning from one to the other caused numerous interoperability issues, resulting in a multi-profile specification to define standard ways of making that transition. That was not a scalable approach. The new approach was that multiple use cases would exist across multiple devices, and the new architecture should support transitions between any of them. That needed a clean sheet of paper design and required developers to look at audio application development differently. But it gives a new lease on life to audio design. However, it means that audio developers need to understand more about how the Core works, as the GAF lets them configure a much larger part of it than was previously possible. So, let's start with that.

ISOCHRONOUS CHANNELS

The basic concept of Isochronous Channels is straightforward. At their simplest, the Bluetooth LE Audio specifications assume two devices. The first, referred to as an

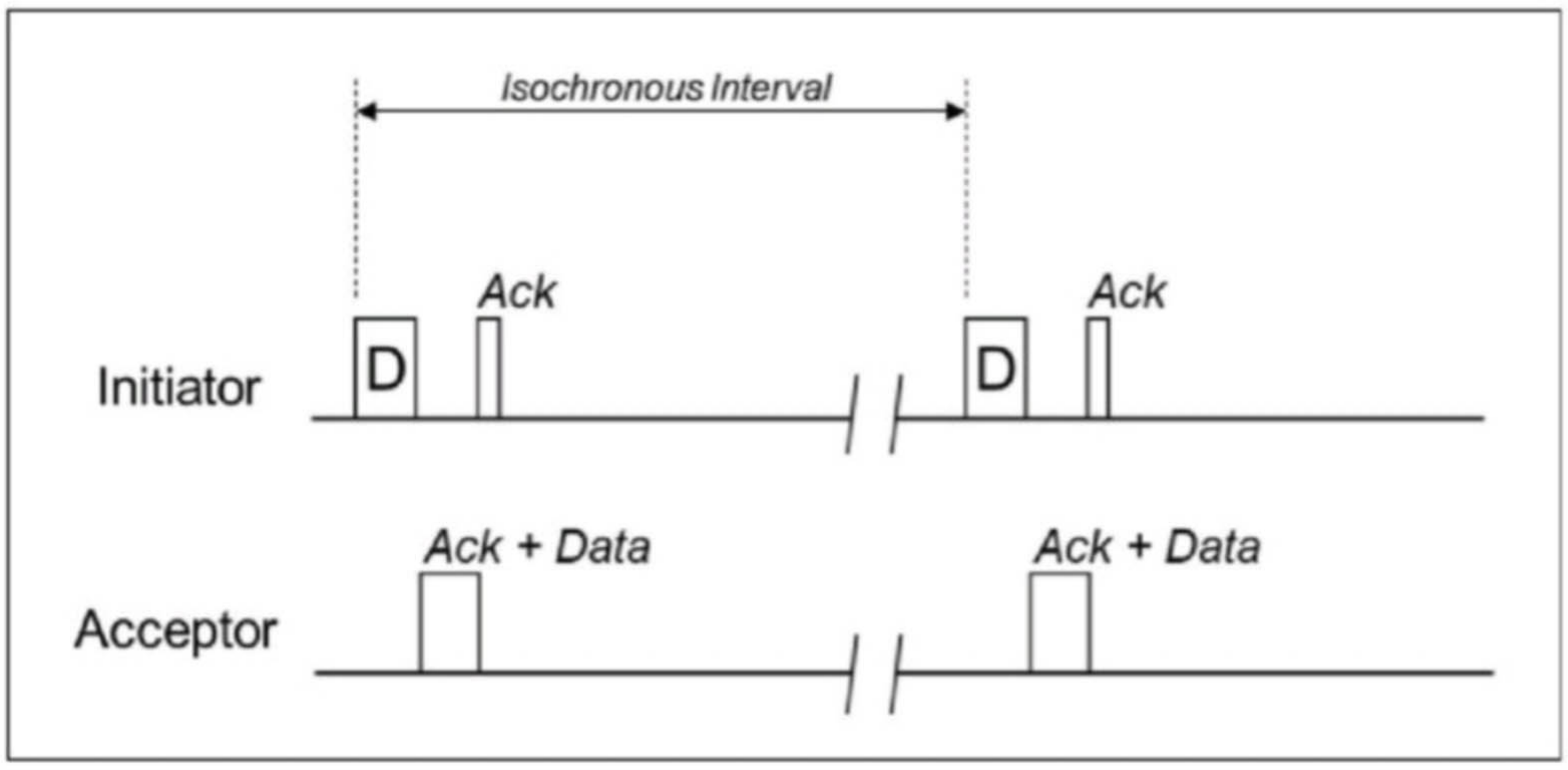


FIGURE 2
The BLE Audio Initiator sends an encoded audio data packet (D) to the Acceptor.

"Initiator," is responsible for configuring an "Acceptor" and then sending and receiving packets containing audio data at a regular interval called the Isochronous Interval, which can be any multiple of 2.5ms. However, the recommended duration is 10ms, as that's the optimum frame length for the new LC3 codec.

Figure 2 shows the Initiator sending an encoded audio data packet (D) to the Acceptor. If it is received, the Acceptor acknowledges it. If the Acceptor is also generating audio data, as it would if it were a headset with a microphone for telephony, it includes that data in its acknowledgment. The Initiator will then acknowledge receipt of that data.

We have to contend with interference in the real world, hence the Initiator can define multiple retransmissions for the audio data. It will continue to transmit its audio packet and listen for any audio data from the Acceptor until it knows that all audio data in that Isochronous Interval has been successfully transmitted and received. Once each device has received its audio data, it can go to sleep until the start of the next Isochronous interval. Every transmission moves to a new frequency channel, based on Bluetooth LE's hopping algorithm, to minimize interference.

The example of Figure 2 shows a bidirectional link, where audio data is transferred in both directions. The same arrangement applies where only a unidirectional audio flow is required, such as media streaming from the Initiator or an Acceptor's audio reception. In all cases, the Initiator is responsible for setting everything up and sending packets at a defined time. For unidirectional applications, the unused directions, from the Initiator to the Acceptor or vice versa, will be sent with null audio data packets.

This configuration is called a Connected Isochronous Stream, more commonly abbreviated to CIS, referring to a single direction. A bidirectional CIS could be configured as two separate streams. The bidirectional figure in Figure 2 is an optimization that combines them into one CIS, saving airtime. The two directions in the diagram can still be configured to have different packet sizes to support different codec configurations: they can also use different radio codings. They can be directed to different applications on the Initiator. For example, the stream from Initiator to Acceptor could be streaming high-quality music. In contrast, the return stream from Acceptor to Initiator could be directed to a voice recognition application using a voice-quality codec configuration. Everything can be configured from the GAF specifications.

The GAF defines any number of CISes between an Initiator and one or more Acceptors as part of a Connected Isochronous Group (CIG). While the CIS defines the packets' timing, the CIG adds more application-specific detail. It defines the order in which packets are sent when there is more than one CIS,

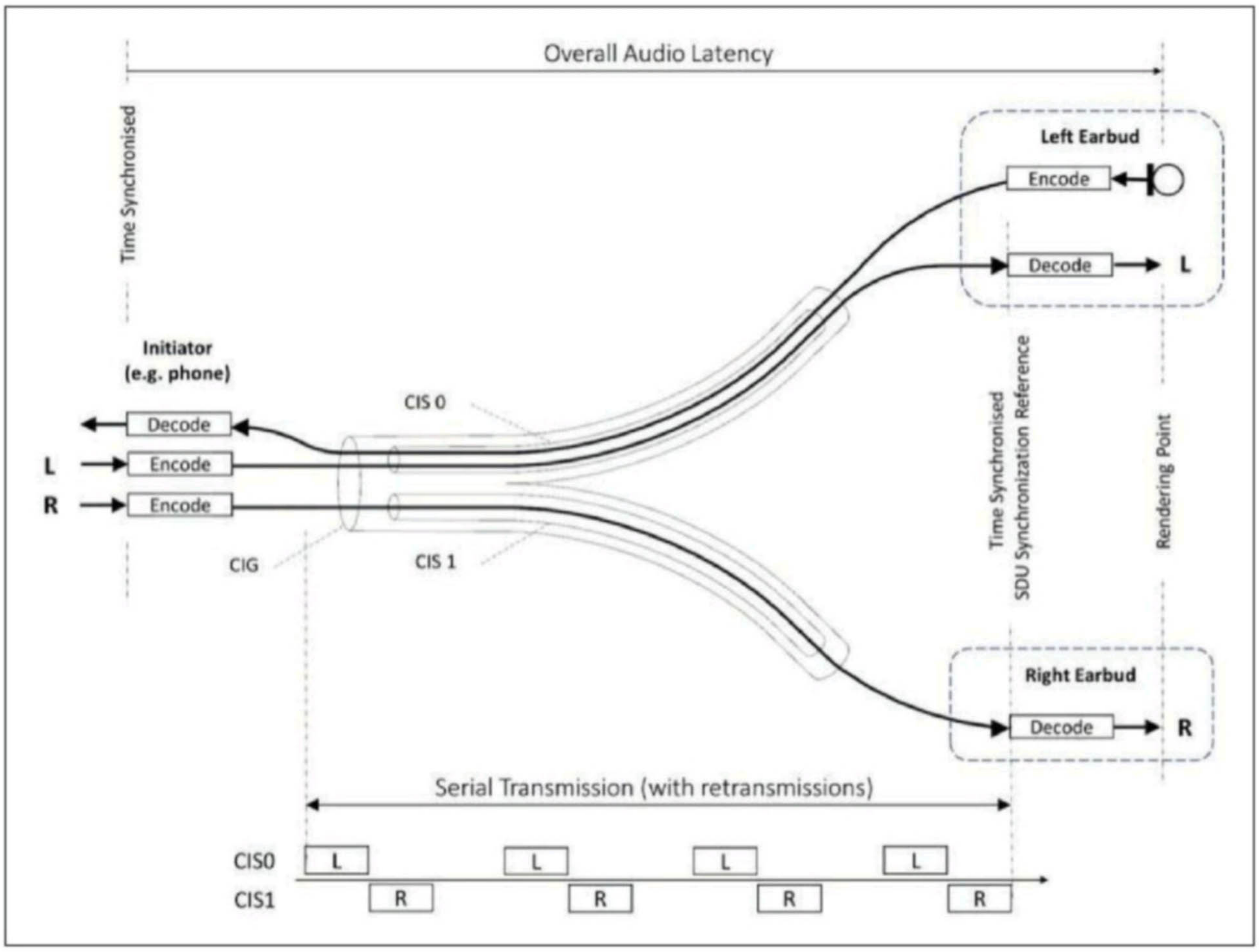


FIGURE 3
The Connected Isochronous Group provides a unique timing reference across all its constituent CISes, synchronizing the audio rendering across different physical devices.

and more importantly, it provides a unique timing reference across all its constituent CISes, which is used to synchronize the audio rendering across different physical devices.

Figure 3 combines these concepts for a pair of stereo earbuds. The left earbud uses its microphone to provide a return audio channel, which might be used for voice recognition, gaming, or telephony.

On the left side of the diagram, the left and right stereo inputs are encoded using the LC3 codec and transmitted in CIS0 to the left earbud and CIS1 to the right earbud. The Left earbud uses its microphone to capture the user's voice and sends it back to the Initiator using the bidirectional CIS0.

At the bottom of the Figure 3, we see the various transmission opportunities and retransmission slots configured by the Initiator. Although the input left and right audio signals are sampled simultaneously, they are sent serially, so the left and right earbuds need to have a common datum to render them simultaneously. As the ear can detect a difference of just a few microseconds, which is how we determine the relative position of a sound, that's an important requirement to include. Incidentally, current TWS earbuds all use proprietary methods to achieve this, so you can't mix and match current devices from different manufacturers.

Bluetooth LE Audio renders left and right audio signals at the same time by defining an SDU reference point that corresponds to the end of the last packet transmitted by the last CIS within a CIG. Each device is then given a delay, called the Presentation Delay, which is usually 10ms to 40ms. Every device must render its audio stream at the end of this delay. Within the Presentation Delay, the rendering device must decode its received audio stream, apply packet loss correction if needed, and perform additional audio processing, such as automatic noise cancellation. This leads to a typical overall audio latency from the input to the outputs of between 25ms and 65ms. The exact value can be negotiated when each CIG is configured.

LC3 CODEC AND LATENCY

The new, mandatory codec, which every Bluetooth LE Audio device must support, is the LC3—the Low Complexity Communications Codec. This modern, efficient, psycho-acoustic codec provides high-quality compression on voice and music. Unlike previous Bluetooth codecs, it is a single-channel codec, so stereo inputs are encoded as separate left and right streams. This is far more efficient for applications like earbuds, as each device only receives the encoded audio data relevant to its operation. This helps to reduce power consumption on earbuds and hearing aids, which generally have much smaller batteries than an Initiator.

LC3 Parameter	Values
Sampling Rate	8kHz, 16kHz, 24kHz, 32kHz, 44.1kHz or 48kHz
Bits per sample	16, 24 or 32
Frame Size	7.5ms or 10ms (The actual size for 44.1kHz sampling is generated automatically when 7.5 or 10ms is set.)
Bytes per frame (payloads per channel)	20 to 400
Number of audio channels	One. The profile may multiplex multiple channels.

TABLE 1
The LC3 codec shows a variety of different sampling rates.

The LC3 is optimized for a frame length of 10ms, providing the best compromise between audio quality and low latency. It is part of the Bluetooth LE Audio specifications, so companies can implement it themselves, although most silicon providers provide it.

The LC3 supports a variety of different sampling rates, as shown in **Table 1**. For example, 16kHz is generally used for voice applications, 24kHz for general-purpose audio, and 48kHz for high-quality audio. In independent, subjective audio tests, participants usually cannot distinguish between 48kHz LC3 sampled music and the original stream, with some preferring the LC3 encoded version. The tests also show that the perceived LC3 quality at any sampling rate is similar to the older SBC codec at double that rate. For example, the quality of current Bluetooth audio products using SBC at 48Hz is similar to that of LC3 encoding at 24kHz. This is important for developers, as pushing up the sampling rate results in larger packets. Larger packets are more vulnerable to interference and take up more airtime. When you add in retransmissions, employing the highest bitrates can use most of the available radio bandwidth. The greater efficiency of the LC3 means that developers have room to

ABOUT THE AUTHOR

For the past 30 years **Nick Hunn** has been closely involved with short range wireless and communications, designing technology that helps to bring mobility to products, particularly in the areas of telematics, M2M, smart energy, IoT, LPWAN, wearables and mobile health. He coined the word "Hearables" and chaired the Bluetooth Hearing Aid working group, leading the development of LE Audio. Nick has published *An introduction to Bluetooth LE Audio* now in its second edition, and is also the author of *The Essentials of Short Range Wireless* a book attempting to explain the application of wireless technologies to product developers. He regularly writes about technology on his blog at www.nickhunn.com. He can be contacted at nick@wifore.com.

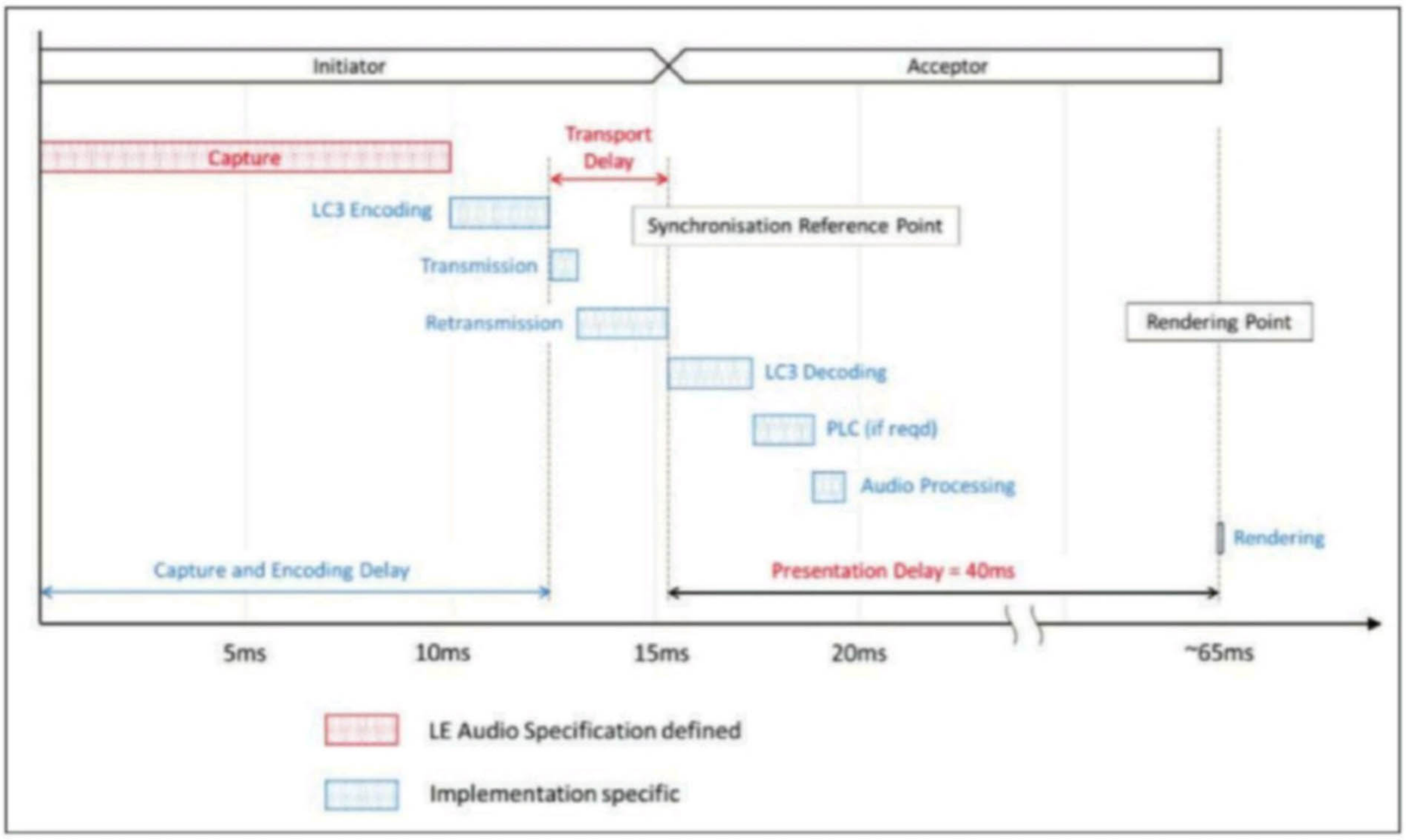


FIGURE 4
Putting all the components of latency together allows developers to determine how to adjust.

maneuver when implementing more complex audio applications or coexisting with other radio traffic in the same device, such as when streaming from a Wi-Fi access point.

Returning to latency, **Figure 4** shows all the different components together to demonstrate how a developer can adjust the overall figure. The items in red are stipulated by various Bluetooth LE Audio specifications; the ones in blue are more under the implementer's control.

At the Initiator, the capture is generally 10ms. There is a 7.5ms option to cope with coexistence with older Bluetooth devices like mice and keyboards, which may use 7.5ms transmission slots, but 10ms is recommended. The following encoding time depends on platform resources and whether LC3 encoding is performed separately or in parallel across multiple input channels.

The choice of codec configuration determines the length of the transmitted packets. As sampling rates are increased from 8kHz to 48kHz, the resulting packet size can expand from 16 octets to 155 octets. The higher layer application also determines the number of retransmissions, although the chip's controller implementation may override that. These choices mean that the transmission time (shown as the Transport Delay) can vary from less than a millisecond to almost ten milliseconds. More input channels will increase that. The Core also allows some very complex transmission schemes spread over multiple Isochronous Intervals, but we'll ignore that complexity for now.

Once the Acceptors receive the audio data packets, they need to start by decoding them. That depends on processing resources but typically only takes a few milliseconds. Packet loss correction may need to be applied, followed by device-specific audio processing. After that, the decoded audio must be buffered before being rendered at the end of the Presentation Delay. Depending on the complexity of the device-specific processing, the decoding and processing of the Acceptor may take anything between 3ms and 40ms. Unicast devices, which use Connected Isochronous Streams, negotiate this value when configuring themselves, resulting in the most appropriate overall audio latency for a particular application. The situation is different for broadcast, as the specification allows the Initiator to set a unilateral value.

That brings us to the end of Part One of the overview of the new Bluetooth LE Audio specifications, such as the Generic Audio Framework, Isochronous Channels, the LC3 Codec, and latency. In Part Two, we cover Broadcast Isochronous Streams, CIS, BIS broadcast transmitter configurations, and other aspects.

Broadcast capabilities transform the topology and audio usage models, particularly those used for the new Auracast applications. They're probably the most significant advance in audio since the introduction of stereo back in the 1950s. With the information in Part Two, developers can think outside the box of conventional audio applications to open up new ways to experience and share sound.

Additional materials from the author are available at:
www.circuitcellar.com/article-materials

Reference [1] as marked in the article can be found there.

RESOURCES

Bluetooth SIG | www.bluetooth.com

FREE Download Now!
Mac, Windows and Linux

Blackmagicdesign

DaVinci Resolve
Media Pool
Aerospace Propulsion Project
Main Edit
01:00:12:17
DaVinci Resolve 19
Media
Cut
Edit
Fusion
Color
Fairlight
Deliver

Edit and color correct using the same software used by Hollywood, for free!

DaVinci Resolve is Hollywood's most popular software! Now it's easy to create feature film quality videos by using professional color correction, editing, audio and visual effects. Because DaVinci Resolve is free, you're not locked into a cloud license so you won't lose your work if you stop paying a monthly fee. There's no monthly fee, no embedded ads and no user tracking.

Editing, Color, Audio and Effects!

DaVinci Resolve is the world's only solution that combines editing, color correction, visual effects, motion graphics and audio post production all in one software tool! You can work faster because you don't have to learn multiple apps or switch software for different tasks. For example, just click the color page for color, or the edit page for editing! It's so incredibly fast!

Professional Editing

DaVinci Resolve is perfect for editing sales or training videos! The familiar track layout makes it easy to learn, while being powerful enough for professional editors. You also get a library full of hundreds of titles, transitions and effects that you can add and animate! Plus, DaVinci Resolve is used on high end work, so you are learning advanced skills used in TV and film.

Creative Color Correction

DaVinci Resolve's color page is Hollywood's most advanced color corrector and has been used on more feature films and television shows than any other system! It has exciting new features to make it easier to get amazing results, even while learning the more advanced color correction tools. There's PowerWindows™, qualifiers, tracking, advanced HDR grading tools and more!

Designed to Grow With You

DaVinci Resolve is designed for collaboration so as you work on larger jobs you can add users and all work on the same projects, at the same time. You can also expand DaVinci Resolve by adding a range of color control panels that let you create unique looks that are impossible with a mouse and keyboard. There's also edit keyboards and Fairlight audio consoles for sound studios!

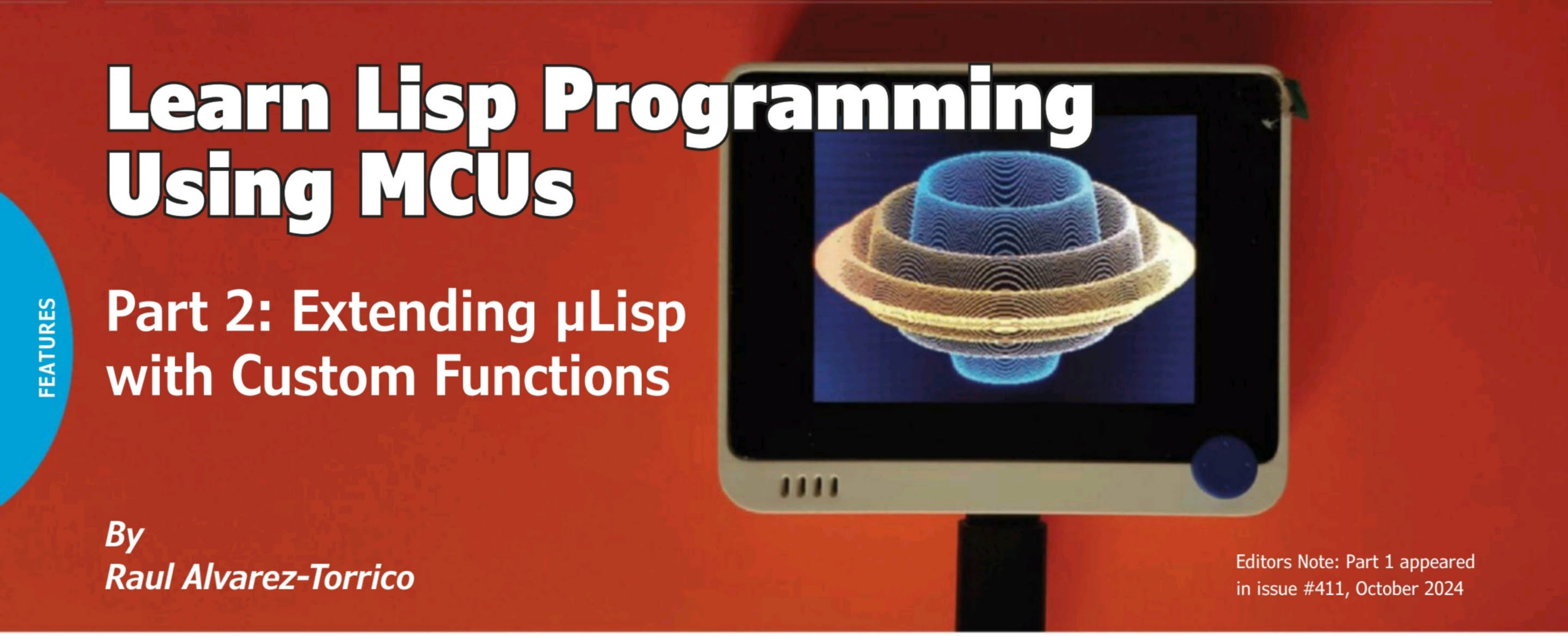
DaVinci Resolve 19	**Free**
DaVinci Resolve Micro Color Panel	**Only $509**

Learn the basics for free then get more creative control with our accessories!

www.blackmagicdesign.com

Learn More!

NO SUBSCRIPTIONS • NO ADS • NO USER TRACKING • NO AI TRAINING



Learn Lisp Programming Using MCUs

Part 2: Extending μLisp with Custom Functions

By
Raul Alvarez-Torrico

Editors Note: Part 1 appeared in issue #411, October 2024

In the second installment of this series, Raul delves into the world of μLisp and shows how to extend capabilities by adding custom functions such as "pulsein" and "servo-create", enabling developers to interface with ultrasonic sensors and control servomotors. In Part 1, Raul presented μLisp concepts and syntax, and extensions specific to MCUs.

μLisp is a dialect of the Lisp programming language. It is tailored to microcontrollers having limited amounts of RAM, and can be run on various platforms—from the Arduino Uno based on the ATmega328, up to the Teensy 4.0/4.1 [1]. To follow the topics discussed here, you need to be acquainted with the Arduino platform and any programming language from the Lisp family. Please see Part 1 of this article series ("Crash Course Offers Insight into Pioneering Language," *Circuit Cellar* 411, October, 2024) for a brief introduction to Lisp programming and the μLisp dialect [2].

With μLisp, it is possible to add your own extension functions, such that they appear as built into the language after re-compiling and flashing the μLisp interpreter to your Arduino board. There's an excellent tutorial on how to do this on the μLisp website [3]. I will build on that tutorial to discuss how to add custom functions. For the examples presented here, I will continue to use Seeed Studio's Wio Terminal MCU development board; but the information discussed also can be applied to other μLisp-supported Arduino boards.

EXTENDING μLISP

In general, any function not supported by default in μLisp can possibly be added as an extension to the language—for instance, hardware access functions for servomotor control. Let's explore a simple example to help us understand how the language extension system works.

Listing 1 shows the Arduino C code for adding the `pulsein` extension to μLisp, which is not available by default. This extension will make it possible to measure digital pulse widths. Five sections of the listing show the μLisp extensions structure. Like the core μLisp interpreter, extensions are also written in Arduino C language. So, any extensions to the interpreter imply basically writing C code. That makes adding them straightforward, using the Arduino C language built-in functions and external libraries.

In the first "Definitions" section in Listing 1 (lines 1–16), the function, `*fn_pulsein(...)` is the C language implementation for the `pulsein` μLisp function we want to add as extension. Once the extension is added, it will be possible to call it from the REPL, for example, with the expression: `(pulsein 3 1)`. The argument "3" is the pin number on which we want to measure an incoming pulse's width. The second argument "1" (for "high") means we want to measure a high-level pulse. Passing "0" (for "low") instead, we will detect and measure a low-level pulse. So, every time the `pulsein` function is called in the μLisp environment, the `fn_pulsein` C function will be executed to provide this functionality.

Line 7 gets the number of arguments passed to `pulsein`, and line 10 makes sure these are exactly two. If that's not the case, an error message is printed with line 14. Next, line 11 gets the first argument value, checks if it's an integer and stores it in the `pin_number` variable. The `checkinteger()` function takes a μLisp integer object as argument and returns a C integer. Line 12 does the same for the second argument, which defines the detection mode —high or low pulse. Going forward, line 13 calls the Arduino "pulseIn" C function passing `pin_number` and `level` as arguments to measure the incoming pulse. The result is stored in the `pulse_width` variable, which is then returned as the measured value with line 15. The function `number()` used here converts the value from a C integer to a µLisp integer object.

Section two of the extension file is "Symbol names," and it has just line 19. Here, a constant char array is defined containing the string that will be used as the symbol or name for the µLisp function, which is `pulsein`.

Section three is "Documentation strings" (lines 21-24). It defines another constant char array storing the function documentation string. Function documentation is accessed in µLisp by calling the `?` function with the function name as argument. For instance, to access the `pulsein` documentation we will type `(? pulsein)` in the REPL, and the interpreter will give back in response the string from lines 22–24. The same applies for other µLisp built-in functions.

Section four is the "Symbol lookup table" (lines 26–29) that maps the aforementioned extension features. Line 28 shows the first and only entry in the table, which is for the `pulsein` function. As you can see, it contains the function name `stringpulsein`, the function pointer `fn_pulsein`, the documentation `docpulsein` and the "minmax" code, which is `0222`.

As explained on the µLisp website, the minmax is an 8-bit code specified as a three digit octal number starting with a zero (which has no meaning). The second digit — "2" in this example — specifies that we are defining a function. Other options for this digit are: "symbols," "special forms," and "tail-recursive special forms." Don't worry about these, because for now we will be defining just functions.

The third digit, which is also "2," specifies the minimum number of function parameters, and the fourth digit, which is also "2," specifies the maximum number of parameters. For `pulsein`, the minimum and maximum number of parameters are the same, because the function will always need two arguments — the pin number and the pulse mode. There could be functions, however, with different number of minimum and maximum arguments to achieve some kind of function overloading.

```
1 // Definitions
2 object *fn_pulsein (object *args, object *env) {
3   (void) env;
4   byte pin_number;
5   bool level;
6   unsigned int pulse_width;
7   int nargs = listlength(args);
8
9   // Get pulse width
10   if (nargs == 2) {
11     pin_number = checkinteger(first(args)); // Get 1st argument
12     level = checkinteger(second(args)); // Get 2nd argument
13     pulse_width = pulseIn(pin_number, level); // optional timeout: pulseIn(pin, value, timeout)
14   } else error2(PSTR("wrong number of arguments"));
15   return number(pulse_width);
16 }
17
18 // Symbol names
19 const char stringpulsein[] PROGMEM = "pulsein";
20
21 // Documentation strings
22 const char docpulsein[] PROGMEM = "(pulsein pin level)\n"
23 "Returns the pulse width measurement in 'pin' by detecting 'level'.\n"
24 "'pin' must be already initializaed with pinmode.";
25
26 // Symbol lookup table
27 const tbl_entry_t lookup_table2[] PROGMEM = {
28   { stringpulsein, fn_pulsein, 0222, docpulsein },
29 };
30
31 // Table cross-reference functions
32 tbl_entry_t *tables[] = {lookup_table, lookup_table2};
33 const unsigned int tablesizes[] = { arraysize(lookup_table), arraysize(lookup_table2) };
34
35 const tbl_entry_t *table (int n) {
36   return tables[n];
37 }
38
39 unsigned int tablesize (int n) {
40   return tablesizes[n];
41 }
```

LISTING 1

Code for adding the "pulsein" extension to µLisp.

Finally, lines 31-41 contain functions for table cross-reference. This section doesn't change when you add new extensions, so for the purpose of the examples we will address here, we can safely ignore them.

Once the extension file is ready, it must be put in the same directory as the "µLisp-arm.ino" or the "µLisp-arm-comments.ino" interpreter Arduino sketch. The extension filename is not important, but it has to have the Arduino ".ino" file extension. After including the extension file, re-compile the interpreter and flash it to your Arduino board. After that, the `pulsein` µLisp function will be available for use.

```
1 // Definitions
2 object *fn_pulsein (object *args, object *env) {
3   // ...Omitted for brevity
4 }
5
6 object *fn_delayus (object *args, object *env) {
7   (void) env;
8   unsigned long delay_time;
9   int nargs = listlength(args);
10
11   // Generate delay in microseconds
12   if (nargs == 1) {
13     delay_time = checkinteger(first(args));
14     delayMicroseconds(delay_time);
15   } else error2(PSTR(“wrong number of arguments”));
16   return 0;
17 }
18
19 // Symbol names
20 const char stringpulsein[] PROGMEM = “pulsein”;
21 const char stringdelayus[] PROGMEM = “delayus”;
22
23 // Documentation strings
24 const char docpulsein[] PROGMEM = “(pulsein pin level)\n”
25 “Returns the pulse width measurement in ‘pin’ by detecting
‘level’.\n”
26 “’pin’ must be already initializaed with pinmode.”;
27
28 const char docdelayus[] PROGMEM = “(delayus time)\n”
29 “Generates a delay of ‘time’ microseconds\n”
30 “using the delayMicroseconds Arduino function.”;
31
32 // Symbol lookup table
33 const tbl_entry_t lookup_table2[] PROGMEM = {
34   { stringpulsein, fn_pulsein, 0222, docpulsein },
35   { stringdelayus, fn_delayus, 0211, docdelayus },
36 };
37
38 // Table cross-reference functions
39 tbl_entry_t *tables[] = {lookup_table, lookup_table2};
40 const unsigned int tablesizes[] = { arraysize(lookup_
table), arraysize(lookup_table2) };
41
42 const tbl_entry_t *table (int n) {
43   return tables[n];
44 }
45
46 unsigned int tablesize (int n) {
47   return tablesizes[n];
48 }
```

LISTING 2
Code additions for the "delayus" extensions.

If you recall from Part 1, I chose Seeed Studio's Wio Terminal development board for my tests. The full source code of all examples described in this article for the Wio Terminal is available on the *Circuit Cellar* Article Code and Files webpage.

To add more extensions to µLisp, you just have to add them on the same extensions file. **Listing 2** is a second version of the previous extensions file that includes the `delayus` extension —also not available by default. Lines 6-17 contain the `*fn_delaus()` C function that implements the `delayus` behavior. The function accepts just one argument (line 12) —the delay time in microseconds (line 13)— and calls the Arduino `DelayMicroseconds` function to generate such delay (line 14). The function name symbol is defined in line 21, its documentation string in lines 28-30 and its lookup table entry is in line 35. Its minmax code is "0211," which corresponds to a function implementation (the "2" digit) with just one argument (the last 11 digits).

ULTRASONIC SENSOR INTERFACE

With these two extensions at hand, now we can measure distances with an ultrasonic sensor using µLisp. **Figure 1** shows the circuit schematic of the HC-SR04 ultrasonic sensor connected to the Wio Terminal. The HC-SR04 is pretty much ubiquitous, and its operation is straightforward. If you have never used it before, there's a link on the Circuit Cellar Article Materials and Resources webpage to a detailed explanation of how the sensor works.

Listing 3 contains the µLisp code to read the sensor. Lines 2-11 define the `ultrasonic` function that accepts two arguments; `trig` is the output pin number to send the trigger signal to the sensor and `echo` is the input pin number to read back the echo pulse. The incoming pulse width contains information about the distance to the detected object. Basically, to obtain a measurement, a 10ms pulse must be provided to the sensor via the "trig" pin, and the sensor will respond back with a high pulse on the "echo" pin. This pulse's width is proportional to the distance traveled by the ultrasonic burst emitted by the sensor's emitter and detected by its receiver —after bouncing back in a nearby object. **Figure 2** illustrates the working principle of the ultrasonic sensor.

Back to Listing 3—line 3 configures pin "trig" as output and line 4 configures pin "echo" as input. Next, lines 5-6 write a low logic level to pin "trig" and generate a 5µs delay to stabilize the output signal. Line 7 generates the trigger pulse ascending edge, the delay in line 8 generates the pulse's 10µs width, and line 9 ends it by generating the descending edge.

Finally, the expression `(pulsein echo 1)` in line 10 calls the `pulsein` function over pin "echo" to detect the width of an incoming high pulse —hence the argument "1." Next, with the expression `(/ (pulsein echo 1) 58.2)` the obtained pulse width is divided by 58.2 to get the distance in centimeters. The 58.2 divisor comes from calculations involving the speed of sound propagating in open air. In a Lisp function such as the one just described, the return value is by default the value of the last expression evaluated inside the function. So, the return value from the `ultrasonic` function will be the value from line 10. To take a distance measurement, copy and paste lines 2-11 to the µLisp REPL, and run line 13 to call the function and obtain a measurement.

Note: When copying µLisp code to the Arduino serial monitor, discard the comments (the text that begins with ";;"), because sometimes they can break the interpreter [4].

Lines 16-29 show a second version of the `ultrasonic` function that prints the measurement to the Wio Terminal's LCD. This version uses a local variable to store the computed distance (see lines 17 and 26), and calls the function `display-distance` —defined in lines 32-39— to display the measurement on the LCD. To use this second version, copy/paste lines 16-39 to the µLisp REPL, and then type line 13 again to run the code. To have access to the LCD display, remember to enable GFX support in the µLisp interpreter Arduino sketch by uncommenting the line `#define gfxsupport` [5]. **Figure 3** shows the ultrasonic example in action.

ADDING SERVOMOTOR SUPPORT

Now let's see how to add support for controlling servomotors. **Listing 4** is an excerpt from the extension code that implements the `servo-create` and `servo-write` µLisp functions. Symbol name definitions, documentation strings, and lookup tables are excluded, because they are similar to what we saw in the previous extension examples. The full source code is available on the Circuit Cellar Article Code and Files webpage. This servo extension uses the "SAMD_ISR_Servo" Arduino library (see line 1), which is specially tailored for SAMD21 and SAMD51 microcontrollers. I used the library's included examples as a base for the extension code. Line 3 defines an array for the pins that will be supported by the extension. For the Wio Terminal, I'm enabling nine pins—0 through 8. Line 4 defines a variable for keeping track of servos already initialized by the `servo-create` function. Lines 6-11

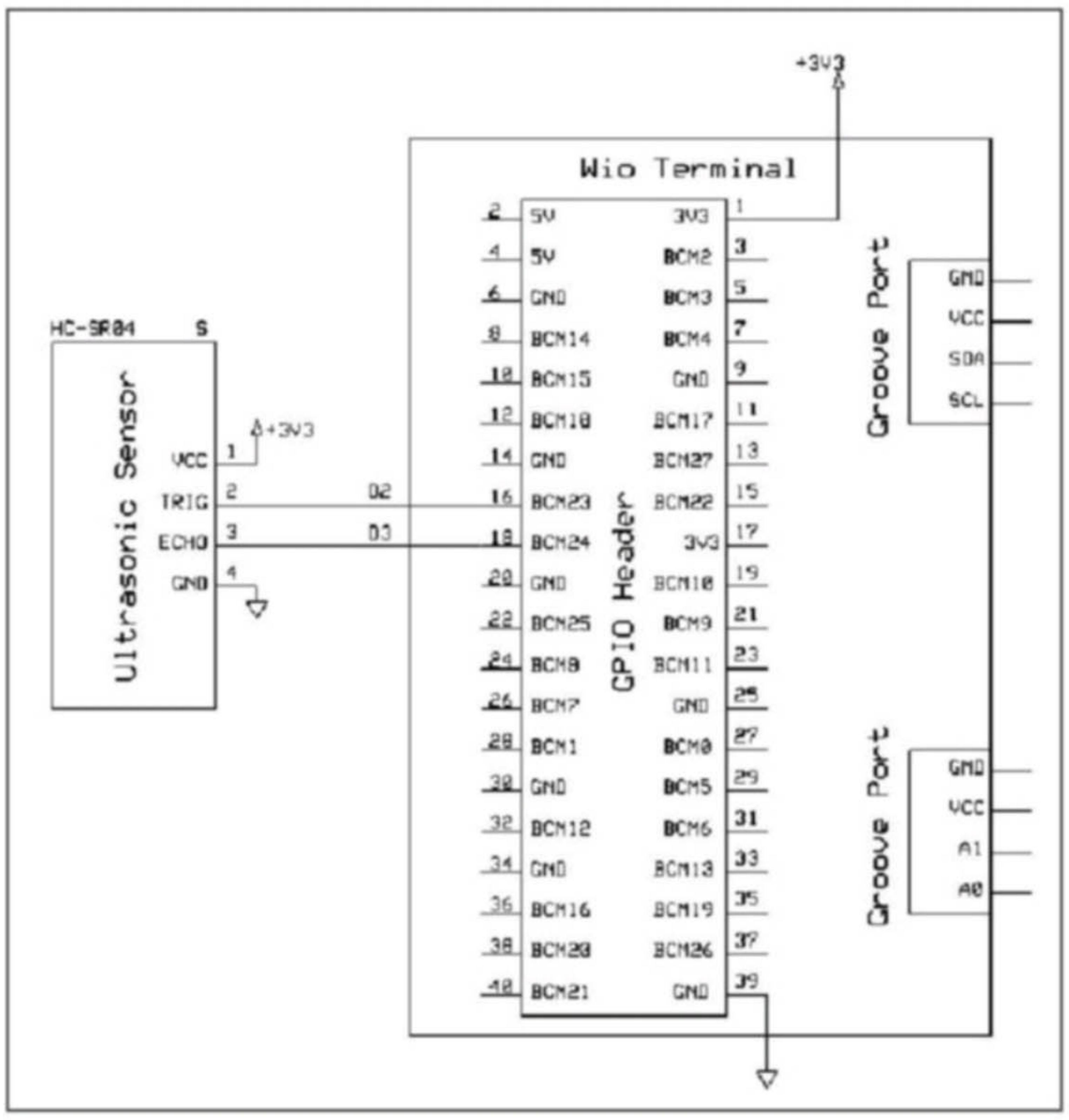

+3V3
Wio Terminal
HC-SR04
S
Ultrasonic Sensor
VCC
TRIG
ECHO
GND
1
2
3
4
+3V3
D2
D3
GPIO Header
2 5V
4 5V
6 GND
8 BCM14
10 BCM15
12 BCM18
14 GND
16 BCM23
18 BCM24
20 GND
22 BCM25
24 BCM8
26 BCM7
28 BCM1
30 GND
32 BCM12
34 GND
36 BCM16
38 BCM20
40 BCM21
3V3 1
BCM2 3
BCM3 5
BCM4 7
GND 9
BCM17 11
BCM27 13
BCM22 15
3V3 17
BCM10 19
BCM9 21
BCM11 23
GND 25
BCM0 27
BCM5 29
BCM6 31
BCM13 33
BCM19 35
BCM26 37
GND 39
Groove Port
GND
VCC
SDA
SCL
Groove Port
GND
VCC
A1
A0

FIGURE 1
Circuit schematic: Wio Terminal with the HC-SR04 ultrasonic sensor.

```
1 ;; Ultrasonic
2 (defun ultrasonic (trig echo)
3   (pinmode trig t)
4   (pinmode echo nil)
5   (digitalwrite trig nil)
6   (delayus 5)
7   (digitalwrite trig t)
8   (delayus 10)
9   (digitalwrite trig nil)
10   (/ (pulsein echo 1) 58.2)
11   )
12
13 (ultrasonic 2 3) ;; Get measurement
14
15 ;; Ultrasonic displaying data on LCD
16 (defun ultrasonic (trig echo)
17   (let (distance)
18     (pinmode trig t)
19     (pinmode echo nil)
20     (digitalwrite trig nil)
21     (delayus 5)
22     (digitalwrite trig t)
23     (delayus 10)
24     (digitalwrite trig nil)
25     ; Measure pulse and compute distance
26     (setq distance (/ (pulsein echo 1) 58.2))
27     (display-dist distance)
28     )
29   )
30
31 ;; Display distance on LCD
32 (defun display-dist (distance)
33   (fill-screen)
34   (set-text-color 2016)
35   (set-text-size 4)
36   (set-cursor 36 32)
37   (with-gfx (stream)
38   (princ (format nil “Distance:~%~% ~a cm” distance) stream))
39   )
```

LISTING 3
Code for reading an ultrasonic sensor with µLisp.

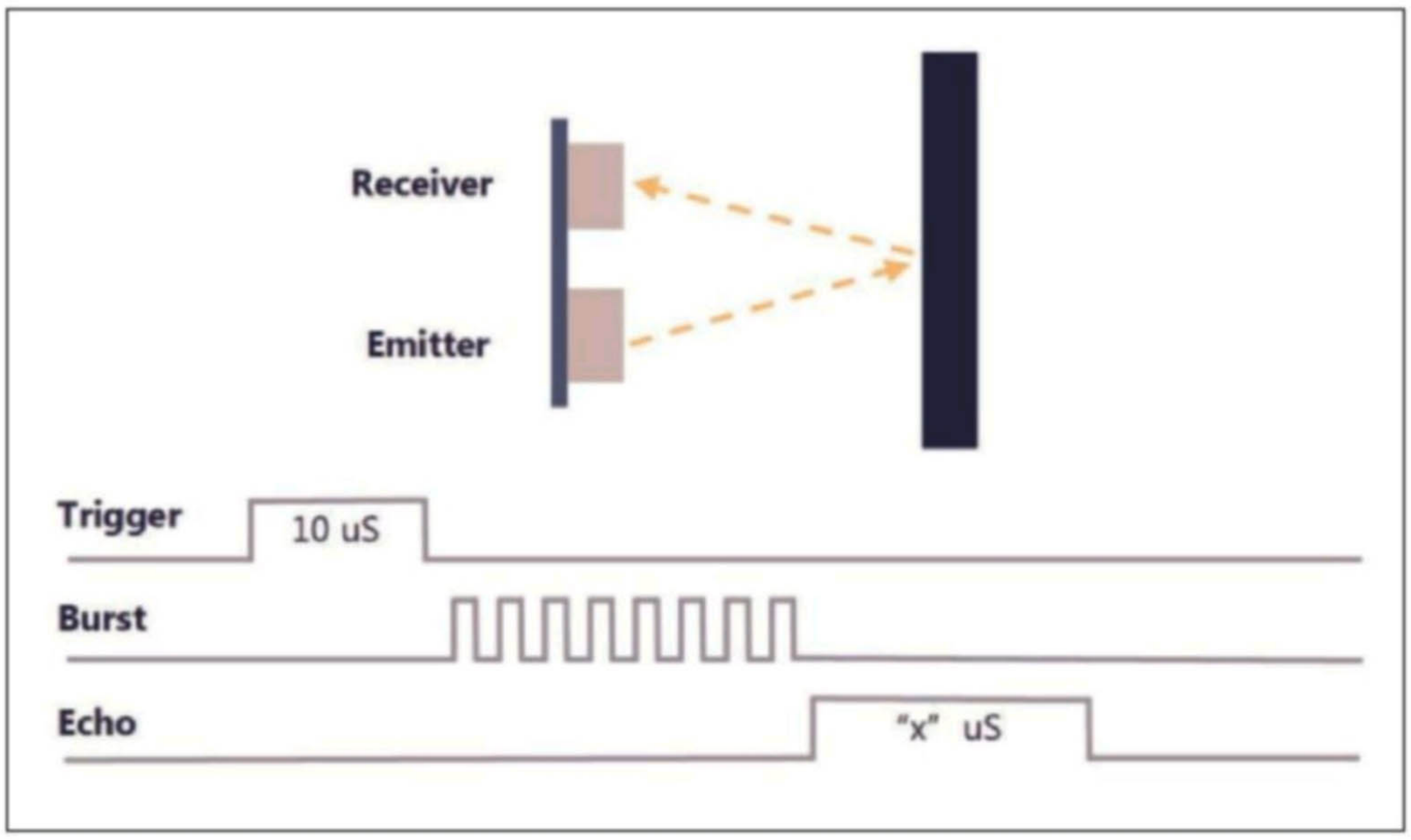


FIGURE 2
HC-SR04 working principle.

define a structure to store the index and assigned output pin for each servomotor. The first field stores the assigned index, and the second field stores its output pin number. Both fields are initialized to "-1" by default.

Next, lines 13-36 implement the `servo-create` function, which receives a pin number as argument. Line 19 retrieves this argument, and line 20 checks if the pin is supported by calling the `Is_Supported(pin_num)` function. The latter function iterates over the `supported_pins` array to check if the pin number in question is in the array. Next, by calling the `Is_Set(pin_num)` function, line 21 checks if the pin has been already set up. To do so, this functions iterates over the `ISR_Servo` array. If the pin in question is not already in use, it will be set up by populating in the structure the current available index (line 22) and pin number (line 23). Next, the pin is hardware-initialized as output (line 24) and set up using the "SAMD_ISR_Servo" library (lines 26-28). Lines 32-34 print error messages to the serial monitor if something goes wrong in the setup procedure. The `Is_Supported` and `Is_set` functions are included in the full source code listing available for download.

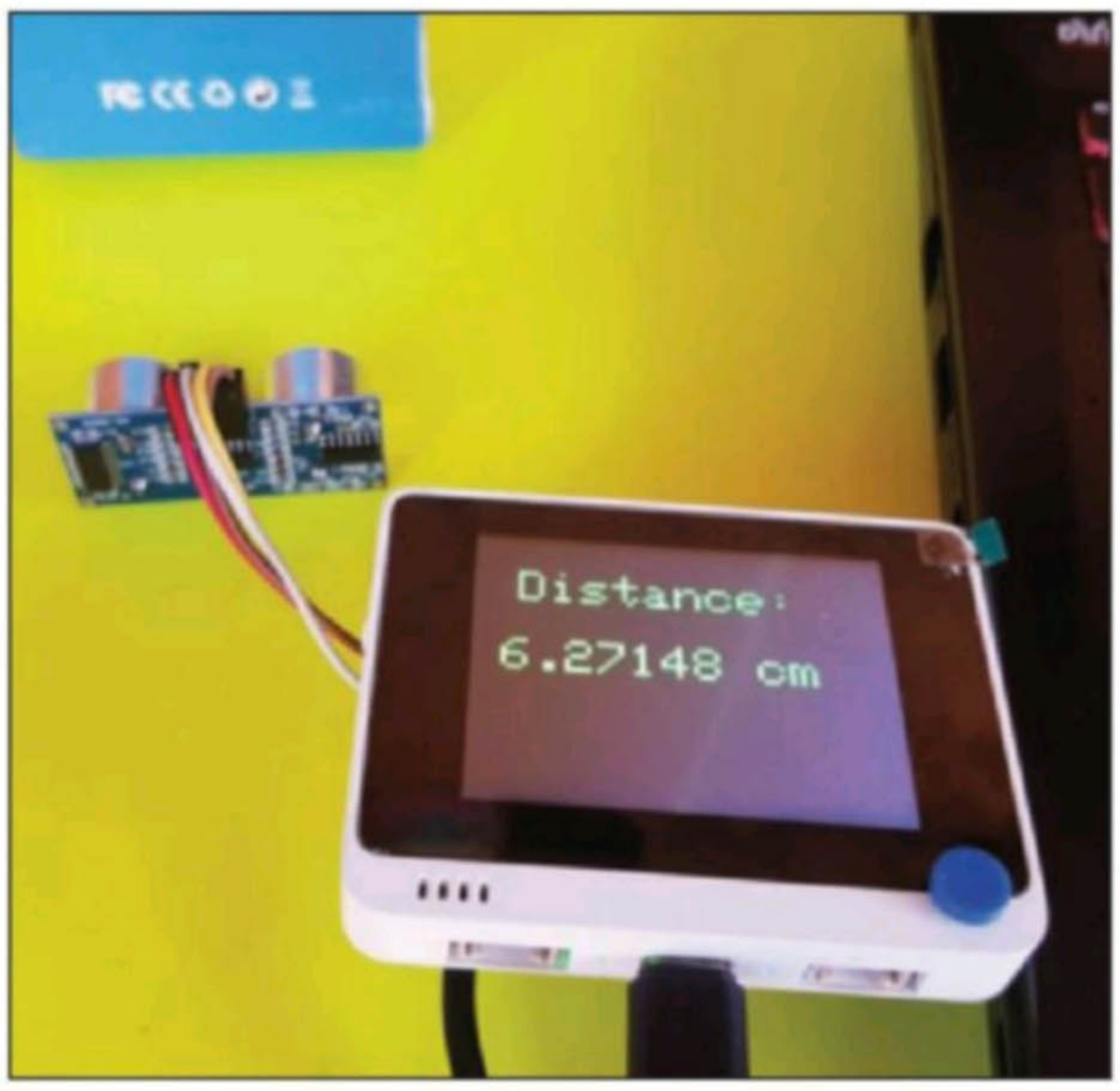


FIGURE 3
Measuring distances using µLisp, the Wio Terminal, and the HC-SR04 ultrasonic sensor.

Lines 38-50 define the `servo-write` µLisp function, which receives two arguments—a pin number and an angle in degrees. Line 43 checks if the function has received the two required arguments. If so, line 44 retrieves the first one which is the pin number. Next, line 45 checks if the specific pin has already been set up. If so, line 46 uses the "SAMD_ISR_Servo" library `setPosition` function to drive the servo to the desired angle. The function call `checkinteger(second(args))` in line 46 retrieves the second argument, which is the angle in degrees.

µLISP CONTROLLED ROBOTIC ARM

Now let's put to a test our newly implemented multi-servo µLisp extension by implementing a simple mini robotic arm with three degrees of freedom (3DOF). This project also will allow us to extend our Lisp knowledge by using some other Lisp tools and techniques. **Figure 4** shows the robotic arm's circuit schematic, and **Listing 5** is the µLisp code that controls it. Let's see how it works.

Lines 1-28 define the function `robo-arm` that controls the robotic arm. Lines 2-3 declare local variables—called lexical variables in Lisp—some of them with initial values. Line 5 uses our newly implemented `servo-create` extension to initialize three servomotors with the pin numbers received as function arguments (see line 1). Lines 6-7 uses our brand-new `servo-write` extension to set the three servos to the default angles defined in line 3. Line 8 initializes pin 6 as an input with its internal pull-up resistor on, which works great for connecting switches. Next, lines 10-27 contain the infinite loop that controls the robotic arm in response to inputs from the joystick module.

The first S-expression in Line 12, `(setq x-in (analogread 4))` performs two tasks. `(analogread 4)` reads the analog signal from input pin 4 connected to the joystick module's VRX output. `(setq x-in (...))` assigns the read value to the `x-in` variable—corresponding to the "x" Cartesian axis.

The second S-expression in line 12 does the same for the "y" axis reading from input pin 5 connected to the joystick's VRY output.

The third S-expression reads the joystick's switch output SW connected to input pin 6. This is a momentary switch activated when the joystick is pressed down. Because the joystick has only two degrees of freedom and we have three servomotors, we will use this signal to switch control between the shoulder and elbow servos, so both can be controlled by the same VRY joystick signal. The VRX signal is permanently assigned to control the base servo. Next, line 13 causes a 20ms delay to debounce the switch input. As you perhaps already know, electromechanical switches must be debounced when connected to MCUs. If you are not familiar with the concept, check the *Circuit Cellar* Article Materials and Resources webpage for suggested readings about switch debouncing.

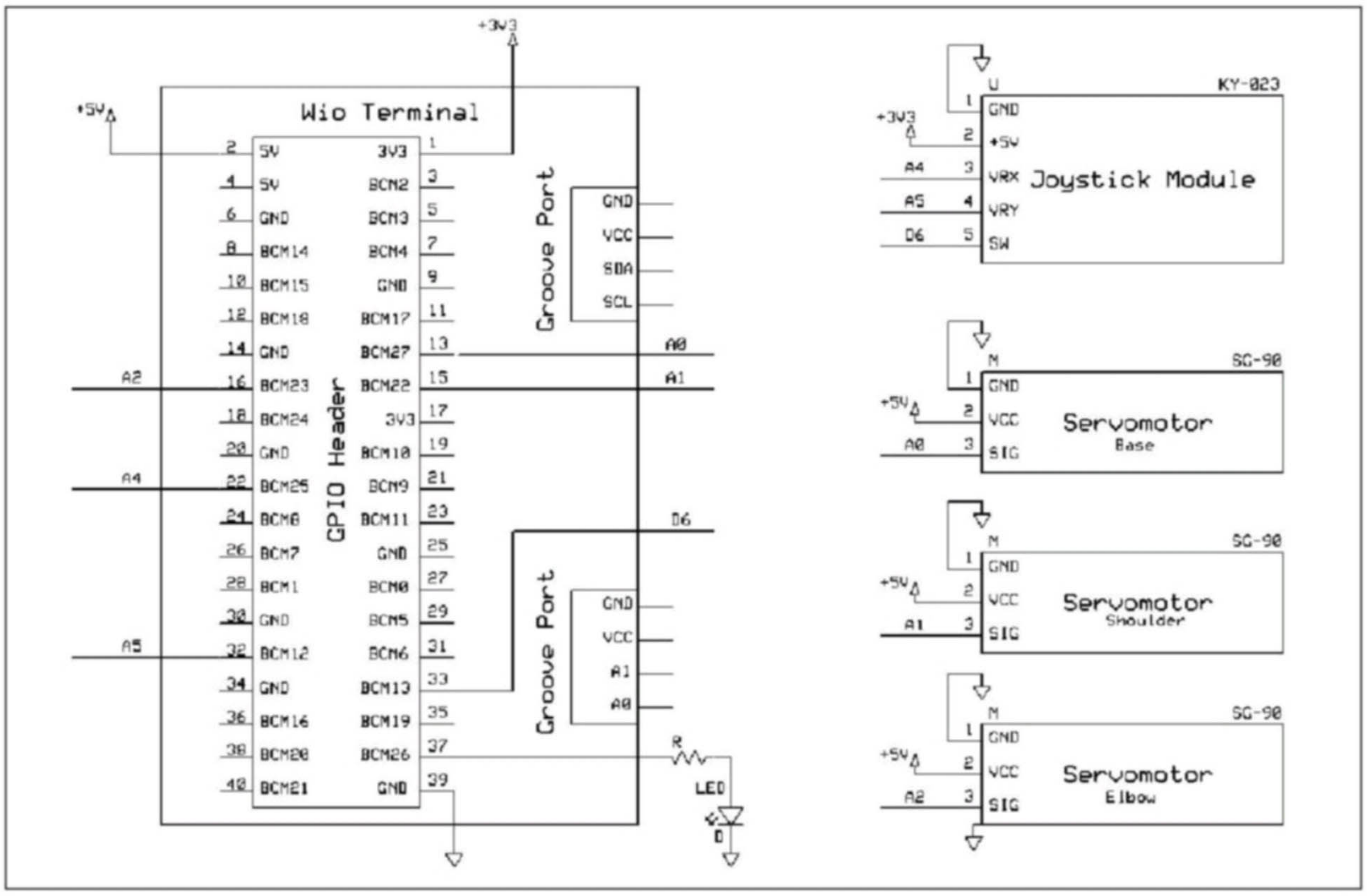
Line 14 performs the debounce verification. The expression `(and (not sw-in) (not (digitalread 6)))` is a Boolean "and" operation to verify two conditions that must occur simultaneously: `(not sw-in)` verifies that the previous switch reading stored in `‘sw-in’` is low. `(not (digitalread 6))` reads the input pin 6 again and verifies that its current value is also low. If these two conditions are both true at the same time, we can assume that the switch has been pressed once. The resulting Boolean value from this verification is then used in the `cond` conditional expression in line 14, which is similar to an "if-then" conditional statement in C/C++ languages. So, in line 14, the expression: `(setq shoulder-on (not shoulder-on))` is executed only if the former "and" predicate is true. If that's the case, the current `shoulder-on` Boolean value will be negated and then re-assigned to itself to change the shoulder/elbow control functionality. The above described switch debouncing is not ideal. There are better ways for sure, but it suits the purpose of keeping our µLisp code simpler.

Line 16 calls the `control-servo` function—defined in lines 31-38—to drive the base servo to a new angle whenever

```
#include “SAMD_ISR_Servo.h”
// Some code lines have been removed for brevity...
byte supported_pins[NUM_SERVOS] = {0, 1, 2, 3, 4, 5, 6, 7, 8};
uint8_t servo_create_idx = 0; //Index to count created servos

typedef struct {
  int     servoIndex;
  uint8_t servoPin;
} ISR_servo_t;
ISR_servo_t ISR_servo[] = {
  {-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}, {-1,-1}, {-1,-1} };

object *fn_servocreate (object *args, object *env) {
  (void) env;
  int pin_num;

  int nargs = listlength(args);
  if (nargs == 1) {
    pin_num = checkinteger(first(args));
    if(Is_Supported(pin_num)) {
      if(!Is_Set(pin_num)) {
        ISR_servo[servo_create_idx].servoIndex = servo_create_idx;
        ISR_servo[servo_create_idx].servoPin = pin_num;
        pinMode(ISR_servo[servo_create_idx].servoPin, OUTPUT);

        SAMD_ISR_Servos.useTimer(USING_SAMD_TIMER);
        SAMD_ISR_Servos.setupServo(ISR_servo[servo_create_idx].servoPin, MIN_MICROS, MAX_MICROS);
        SAMD_ISR_Servos.setReadyToRun();

        ++servo_create_idx;
      }
      else Serial.println(“Pin used”);
    } else error2(PSTR(“pin not supported”));
  } else error2(PSTR(“wrong number of arguments”));
  return 0;
}

object *fn_servowrite (object *args, object *env) {
  (void) env;
  int pin_num;

  int nargs = listlength(args);
  if (nargs == 2) {
    pin_num = checkinteger(first(args));
    if(Is_Set(pin_num)) {
      SAMD_ISR_Servos.setPosition(pin_num, checkinteger(second(args)));
    } else { error2(PSTR(“that servo doesn’t exist!”)); }
  } else error2(PSTR(“wrong number of arguments”));
  return 0;
}
```

LISTING 4
Code for adding servomotor extensions to µLisp.

FIGURE 4
Robotic arm circuit schematic.


+3V3
+5V
Wio Terminal
GPIO Header
2 5V
4 5V
6 GND
8 BCM14
10 BCM15
12 BCM18
14 GND
16 BCM23
18 BCM24
20 GND
22 BCM25
24 BCM8
26 BCM7
28 BCM1
30 GND
32 BCM12
34 GND
36 BCM16
38 BCM20
40 BCM21
3V3 1
BCM2 3
BCM3 5
BCM4 7
GND 9
BCM17 11
BCM27 13
BCM22 15
3V3 17
BCM10 19
BCM9 21
BCM11 23
GND 25
BCM0 27
BCM5 29
BCM6 31
BCM13 33
BCM19 35
BCM26 37
GND 39
A2
A4
A5
A0
A1
D6
Groove Port
GND
VCC
SDA
SCL
Groove Port
GND
VCC
A1
A0
R
LED
D
U
KY-023
+3V3
1 GND
2 +5V
A4 3 VRX
A5 4 VRY
D6 5 SW
Joystick Module
M
SG-90
+5V
1 GND
2 VCC
A0 3 SIG
Servomotor
Base
M
SG-90
+5V
1 GND
2 VCC
A1 3 SIG
Servomotor
Shoulder
M
SG-90
+5V
1 GND
2 VCC
A2 3 SIG
Servomotor
Elbow

there's input from the joystick. The function receives the following arguments: the analog input value "x-in," the :"low-thres" lower threshold value defined as "250" in line 2. This means that the servo will decrease its angle only when an analog value less than 250 is detected. The SAMD51 MCU in the Wio Terminal has an analog-to-digital converter (ADC) of 10 bits. So, the range of discrete values from the analog input is between 0 and 1,023. That means the lower threshold value is located at the top of the lower fourth of that range.

The third argument to the function is the higher threshold defined as "750." When the analog input is greater than this threshold, the servo will start to increase its angle. The fourth argument "base" is the servo pin number, the fifth argument "base-ang" is the destination angle, and the final "dang" argument is the delta that's added/subtracted from the current angle value to increase/decrease it.

As I said before, both the shoulder and elbow servomotors are controlled by the "y" axis potentiometer. Line 18 checks which functionality is currently on. If "shoulder-on" is true, the shoulder servo will be controlled. So, line 20 calls `control-servo` passing the "y-in" ADC value, the "shoulder" servo pin number, the "shoulder-ang" destination angle and the delta "dang". Conversely, if "shoulder-on" is false, the elbow servo will be controlled with line 22 in the same fashion. Note that line 20 is the "then" branch of the "if" statement in line 18 and line 22 is the "else" branch. Next, lines 25-26 print some variable values to the REPL for debugging, and line 27 generates a 200ms delay to avoid cluttering the REPL.

Finally, let's see how the `control-servo` function works. We already explained implicitly the meaning of the six parameters in line 31. Line 33 is an "if-else-if" style conditional statement that checks first if the analog input value is below the lower threshold with the expression `(< analog-in low-thr)`. If that's the case, the expression `(- angle da)` will be executed reducing the current angle by the delta value "da." and then this new value will be set as the new current angle —with the expression, `(setq angle (...))` in the same code line. Line 34 is the "else-if" branch that checks for the higher threshold in a similar fashion. Conversely, it will increase the angle by "da" if the analog input is above the higher threshold.

Going forward, line 36 has two S-expressions. The first one `(setq angle (min angle 180))` limits the newly modified angle to 180° in case the new value exceeds 180. The expression: `(min angle 180)` chooses the minimum value between "angle" and 180. Then, the expression: `(seq angle (...))` assigns this value to the same `angle` variable. The second expression to the right, `(setq angle (max angle 0))` does the same to keep the new angle positive or at least zero. All this is to keep the angles inside the servomotor's working range of 0–180°.

Finally, line 37 moves the servo to its new angle, and line 38 returns the angle to the function caller. For instance, this returning value is stored in line 16 of the `robo-arm` function in the `base-ang` variable with the expression, `(setq base-ang (control-servo ...))`.

Figure 5 shows the robotic arm prototype I built to test the code. The Wio Terminal's GPIO header is compatible with

ABOUT THE AUTHOR

Raul Alvarez-Torrico has a BEng in electronics and is the founder of TecBolivia, a company offering services in physical computing and educational robotics in Bolivia. In his spare time, he likes to experiment with wireless sensor networks, robotics, and artificial intelligence. He is also committed to publishing articles and video tutorials about embedded systems and programming in his native language (Spanish), at their company's site www.TecBolivia.com. You may contact him at raul@tecbolivia.com.

Additional materials from the author are available at:
www.circuitcellar.com/article-materials
References [1] to [5] as marked in the article can be found there.

```
(defun robo-arm (base shoulder elbow)
  (let ((x-in) (y-in) (sw-in) (low-thres 250) (hi-thres 750) (dang 2)
    (shoulder-on t) (base-ang 90) (shoulder-ang 90) (elbow-ang 90))
    ;; Initialize servos
    (servo-create base) (servo-create shoulder) (servo-create elbow)
    (servo-write base base-ang) (servo-write shoulder shoulder-ang)
    (servo-write elbow elbow-ang)
    (pinmode 6 :input-pullup) ;; Initialize joystick switch input
    ;; Infinite loop
    (loop
      ;; Read joystick analog & digital inputs
      (setq x-in (analogread 4)) (setq y-in (analogread 5)) (setq sw-in (digitalread 6))
      (delay 20) ;; Delay to debounce sw-in
      (cond ((and (not sw-in) (not (digitalread 6))) (setq shoulder-on (not shoulder-on))))
      ;; Control base servo
      (setq base-ang (control-servo x-in low-thres hi-thres base base-ang dang))
      ;; Control shoulder or elbow
      (if shoulder-on
        ;; Control shoulder servo
        (setq shoulder-ang (control-servo y-in low-thres hi-thres shoulder shoulder-ang dang))
        ;; Else control elbow servo
        (setq elbow-ang (control-servo y-in low-thres hi-thres elbow elbow-ang dang))
      )
      ;; Debug print to REPL
      (princ (format nil “x-in:~d y-in:~d base-ang:~d shoulder-ang:~d elbow-ang:~d shoulder-on:~d~%”
                x-in y-in base-ang shoulder-ang elbow-ang shoulder-on))
     (delay 200)) ;; To avoid flooding the REPL
    ))

;; Control servo with pot input
(defun control-servo (analog-in low-thr hi-thr servo angle da)
  ;; Decrement or increment angle
  (cond ((< analog-in low-thr) (setq angle (- angle da)))
        ((> analog-in hi-thr) (setq angle (+ angle da))))
  ;; Limit resulting angle to the range 0-180
  (setq angle (min angle 180)) (setq angle (max angle 0))
  (servo-write servo angle) ;; Move servo to angle
  angle) ;; Return new angle value

;; Run code. Press ‘~’ + <Enter>  to end
(robo-arm 0 1 2)
```

LISTING 5
Code for robotic arm.

the Raspberry Pi GPIO header. That allowed me to use the “Pi T-Cobbler” breakout board to connect the signals using a breadboard.

CONCLUSION

In this part of the article series on µLisp, I have shown you how to extend µLisp by including functions not supported by default. Doing so is easy if you already know how to program in C/C++ and you are familiar with the Arduino platform. I also added support for interfacing an ultrasonic sensor and interfacing servomotors.

The servomotor extensions let us showcase µLisp code to control a basic 3DOF robotic arm using a joystick. This, in turn, highlighted a few more quirks in Lisp programming. In the upcoming last part of this article series, Part 3, I will introduce a couple more advanced Lisp programming concepts and use all that we already know about µLisp to present more advanced project examples. Stay tuned!

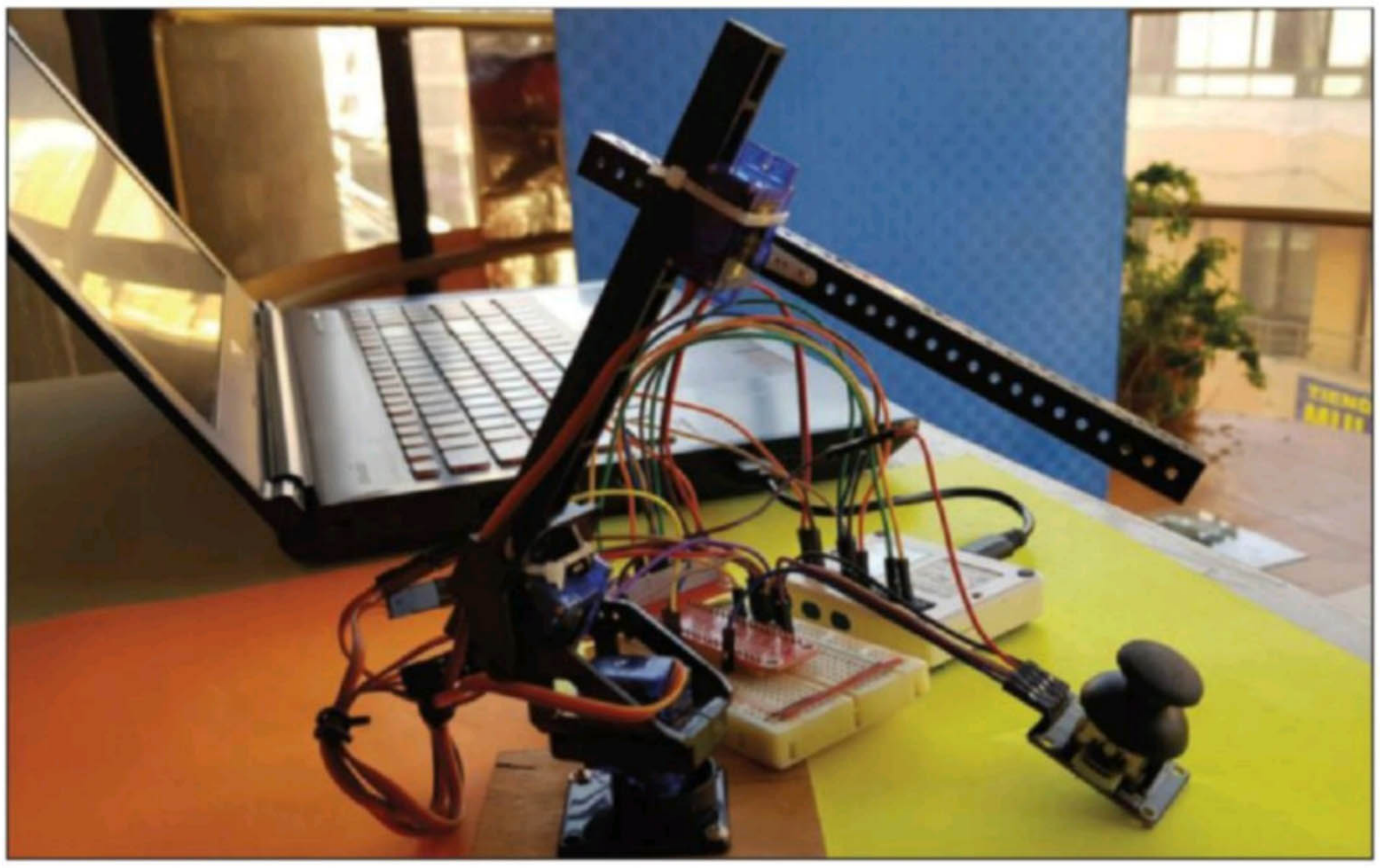


FIGURE 5
Robotic arm testing prototype.

Reviving Frogger's Retro Audio with a Programmable Sound Generator

Using an AY-3-8910 Chip and a Z80 Emulator on a Raspberry Pi

By
Chris Cantrell

The Frogger arcade video game is considered one of the best of all time. Chris shares his journey to bring the nostalgic sounds of Frogger back to life using the classic AY-3-8910 sound chip and a modern Raspberry Pi setup with coding insights and a fix to the original game's music glitch.

I have become that guy—the old man at the office who bores the young engineers with tales of how things used to be. DigiKey? The Internet? Back in my day (the early 1980s), we rode our bikes to RadioShack and browsed racks of electronic components by hand—resistors, capacitors, breadboards, LEDs—you name it. I remember the Forrest Mims engineering notebooks on a shelf as I walked in. These were our definitive guides to electronics, op amps, and logic gates. You can still read them on the Web today. Next to the bookshelf was the wall of chips—analog and digital ICs in cardboard packaging, with transparent plastic bubbles so you could see the parts before you bought them. And don't forget to use your battery-of-the-month-club card to get a free battery when you check out.

It was on that wall of chips that I first met my old friend, the AY-3-8910 programmable sound generator. It's a 40-pin DIP chip, packed with features: three voices, a noise generator, and a complex envelope control. Over the years, it has been with me on many projects, including an LED movie project I wrote about in *Circuit Cellar* #239, June 2010 [1].

Here I am with another retro project featuring the sound chip. In this article, I'll show you how I used Python to talk to my AY-3-8910 from a Raspberry Pi. That isn't revolutionary; you'll find many hardware and software examples on the Internet. But my project has a cool twist.

I'll show you how I ran the original binary code from the Frogger arcade machine in a Z80 emulator on my Pi. For readers who weren't around in the 1980s, Frogger is an action game, the objective of which is to get several little frogs home safely—by keeping them from drowning in a river, or becoming road kill, or the lunch of a snake or alligator. (You can still buy "refurbished" original Frogger games on Amazon, and can even play it as a board game, on PS1, and online.)

With a bit of Python glue, the emulator talks to the real sound chip to reproduce the original Frogger sounds. I used an Adafruit MacroPad USB keyboard as a soundboard to inject sound commands into the running emulator. You press buttons on the keypad to play the different songs and effects from the Frogger game.

When I disassembled the Frogger ROM, I discovered a bug in the Z80 code. The second voice of the main song cuts out halfway through, because of a glitch in the data. Since this is a fun retro project, I'll show you some of the Frogger Z80 code and how I fixed the music data, so you can hear the main song from Frogger as it was intended, complete for the first time!

AY-3-8910

If you are older, like me, you remember the AY-3-8910. If you've ever been to a video game arcade or museum, you've heard the chip in action in many of your favorite games. Were you a Gyruss fan? It has five AY-3-8910 chips. Time Pilot and TRON each have two. Frogger, even with all its music and sound effects, has just one. Early home computers like Intellivision, Amstrad CPC, and ZX Spectrum also used the chip.

The AY-3-8910 programmable sound generator produces three independent sound channels. You control the sound chip through its 16 registers in a two-step process. First, you write the register address (0 through 15) to the chip's 8-bit data bus. Then, you read or write the register's value from the same bus.

The data manual for the chip [2] is a 50-page book chock-full of information, diagrams, and examples. On cold winter nights, nerds like me grab a cup of hot cocoa, pull out the manual, and read down memory lane. Here is a quick summary.

- Each channel has a square-wave tone generator with a 12-bit frequency value. This value is split between two registers for each voice: the fine and coarse tone periods—register pairs 0-1, 2-3, and 4-5.

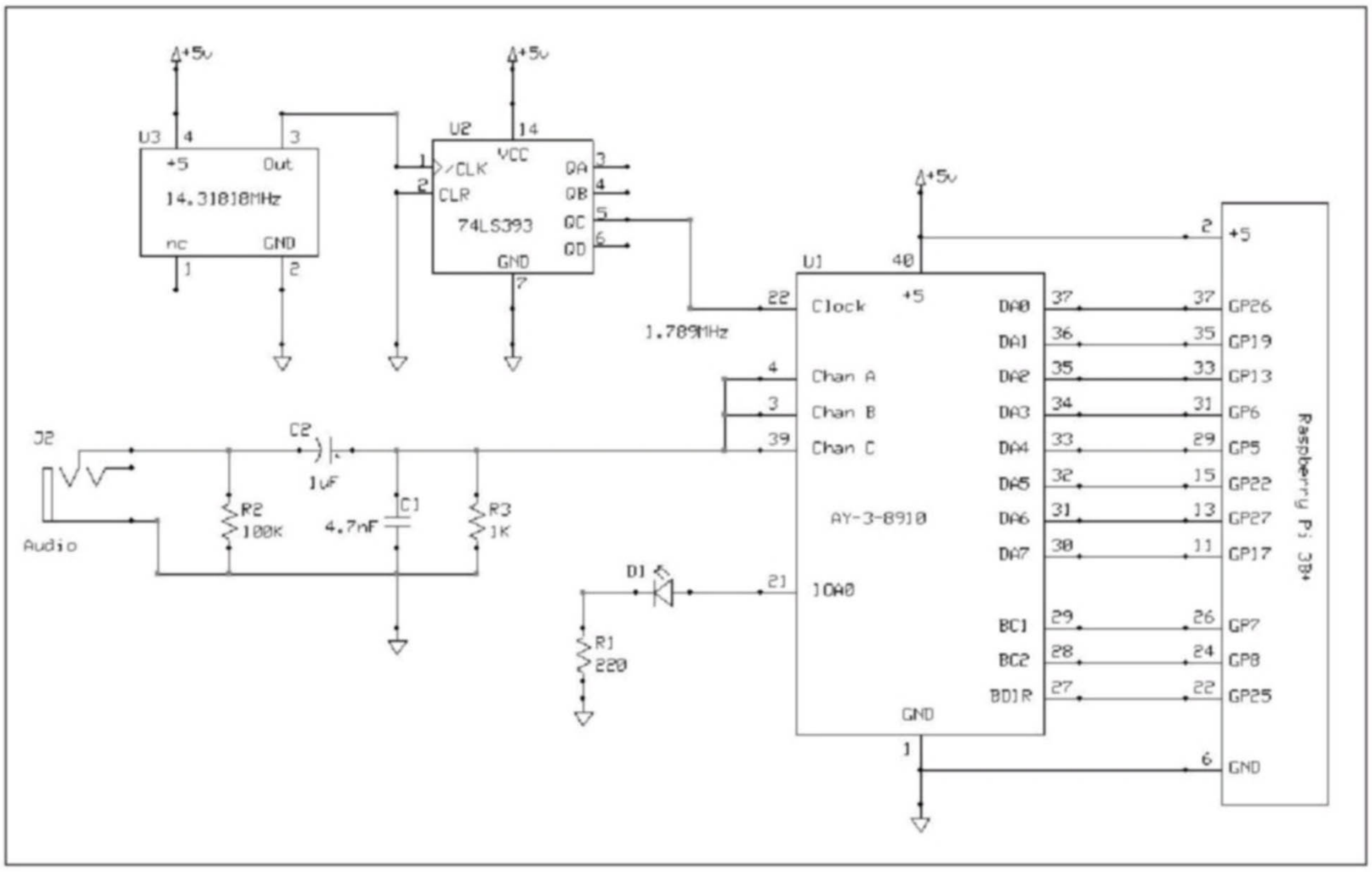

+5v
U3
14.31818MHz
Out
nc
GND
U2
74LS393
VCC
>CLK
CLR
QA
QB
QC
QD
GND
1.789MHz
U1
Clock
Chan A
Chan B
Chan C
AY-3-8910
IOA0
DA0
DA1
DA2
DA3
DA4
DA5
DA6
DA7
BC1
BC2
BDIR
GND
J2
Audio
C2
1uF
R2
100K
C1
4.7nF
R3
1K
D1
R1
220
+5
GP26
GP19
GP13
GP6
GP5
GP22
GP27
GP17
GP7
GP8
GP25
GND
Raspberry Pi 3B+

FIGURE 1
Project schematics, showing how I wired the AY-3-8910 to the Raspberry Pi's GPIO pins. The three audio signals are ORed to an RC filter to clean up the sound. The divide-by-eight binary counter is straight from the Frogger schematics.

- There is a single noise generator with a 5-bit base frequency. This single noise source can be mapped onto one or more of the three channels along with the tones. But again, it is a single noise source. It sounds the same, no matter how many voices mix it in. The noise frequency is configured in register 6. Register 7 is the mixer enabling tones and/or noise on each voice.
- You can assign each channel a 4-bit volume, and change the volume manually over time with your code. The three volume registers are 8, 9, and 10. Or you can use an automatic volume (envelope) controller built into the AY-3-8910. You configure the envelope controller for attacks, decays, and loops. Then you assign it to one or more of the voices. The sound chip changes the volume over time without tying up your processor. But once again, there is only one of these generators. Every voice that uses it shares the same volume pattern. The envelope generator is configured with registers 11, 12, and 13.
- The last two registers, 14 and 15, are general-purpose, 8-bit I/O registers. The bits of these registers connect to TTL-compatible I/O pins on the AY-3-8910. You can use these two ports, ports A and B, to communicate with the outside world—reading DIP switches, blinking LEDs, or receiving sound commands from a master processor.

HARDWARE

The schematics for the project are shown in **Figure 1.** I wired the components on a solderless breadboard, which looks natural for this retro project. **Figure 2** shows the final project. I used short male/female jumper wires [3] to connect the Raspberry Pi's GPIO pins to the sound chip—eight for the databus, and three for the control lines.

The AY-3-8910 chip is nearly 50 years old, but it is easy to buy online. If you don't have one in your collection, try eBay. I bought a couple from Amazon.com [4].

The AY-3-8910 runs on 5V, but many modern micros, like my Raspberry Pi, are 3.3V.

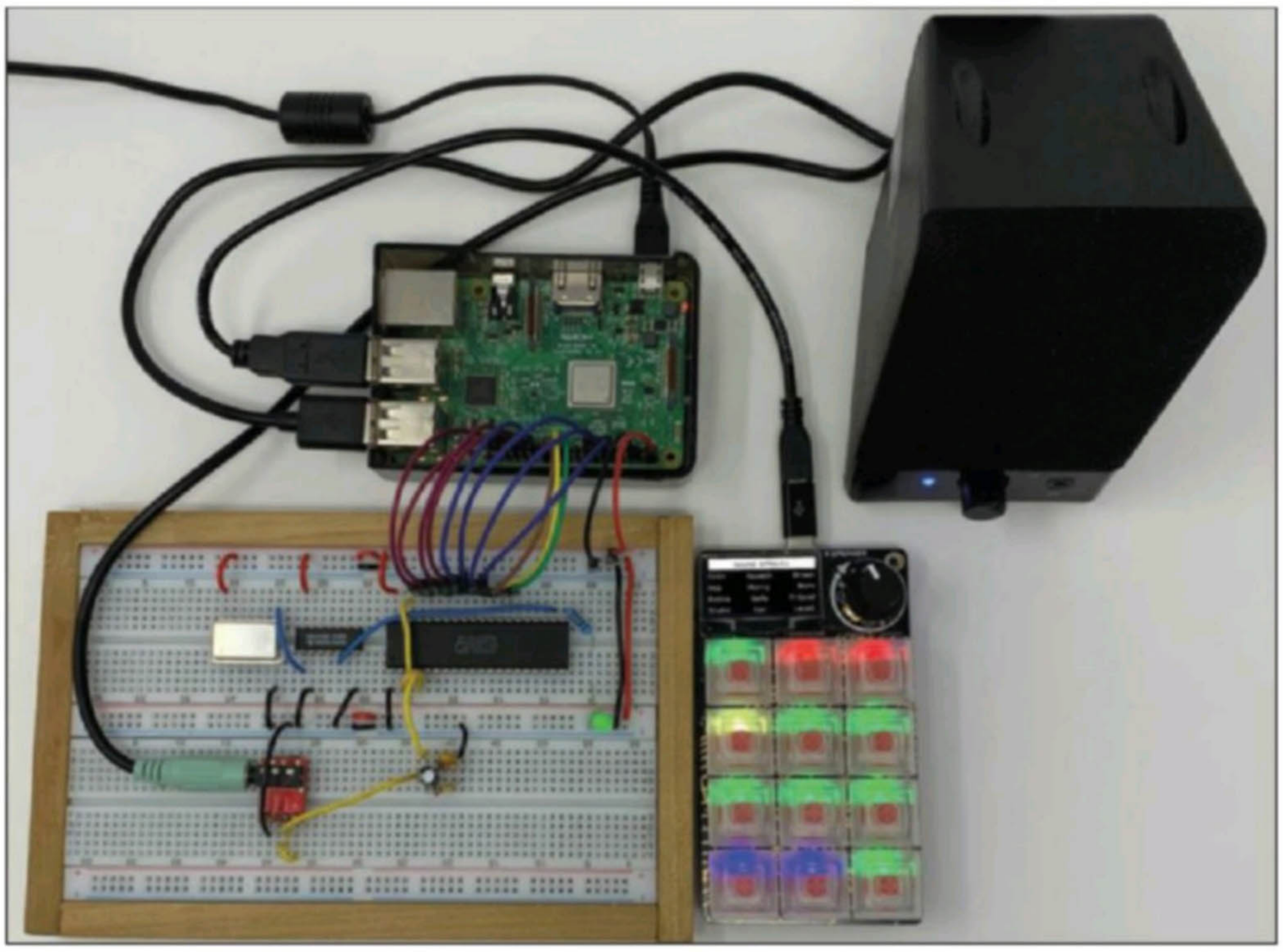


FIGURE 2
Complete Frogger sound project. The Raspberry Pi 3B+ in the top left of the photo is wired to the sound chip on the breadboard in the lower left. There is a single USB-powered speaker in the upper right. The colorful Adafruit MacroPad in the lower right is a USB keyboard that selects the Frogger music and sound effects.

```
# WriteToAY:
# ; A is the address, B is the value
# 0008: D3 80    OUT   ($80),A   ; Write the address to the AY
# 000A: 78       LD    A,B       ; Write the ...
# 000B: D3 40    OUT   ($40),A   ; ... value to the AY
register
# 000D: C9       RET             ; Done

def write_data_pins(self, value):
        bn = 1
        for pin in self.pins_data:
            GPIO.output(pin, bool(value & bn))
            bn = bn << 1

def write_register(self, address, value):
    self.write_data_pins(address)   # Address onto data bus
    GPIO.output(self.pin_bc1, 1)    # BC1:BC2:BDIR = 1:0:0
LATCH
    GPIO.output(self.pin_bc1, 0)    # BC1:BC2:BDIR = 0:0:0
idle
    self.write_data_pins(value)     # Value onto data bus
    GPIO.output(self.pin_bc2, 1)    # BC1:BC2:BDIR = 0:1:0
idle
    GPIO.output(self.pin_bdir, 1)   # BC1:BC2:BDIR = 0:1:1
WRITE
    GPIO.output(self.pin_bdir, 0)   # BC1:BC2:BDIR = 0:1:0
idle
    GPIO.output(self.pin_bc2, 0)    # BC1:BC2:BDIR = 0:0:0
idle
```

LISTING 1
The Frogger hardware maps the AY-3-8910 as two ports: an address latch and a data latch. The comments at the top of the listing show the Z80 routine for writing to the chip. My Python code bit-bangs the three AY-3-8910 control lines (BC1, BC2, and BDIR). The comments in the code show the chip's state at each step.

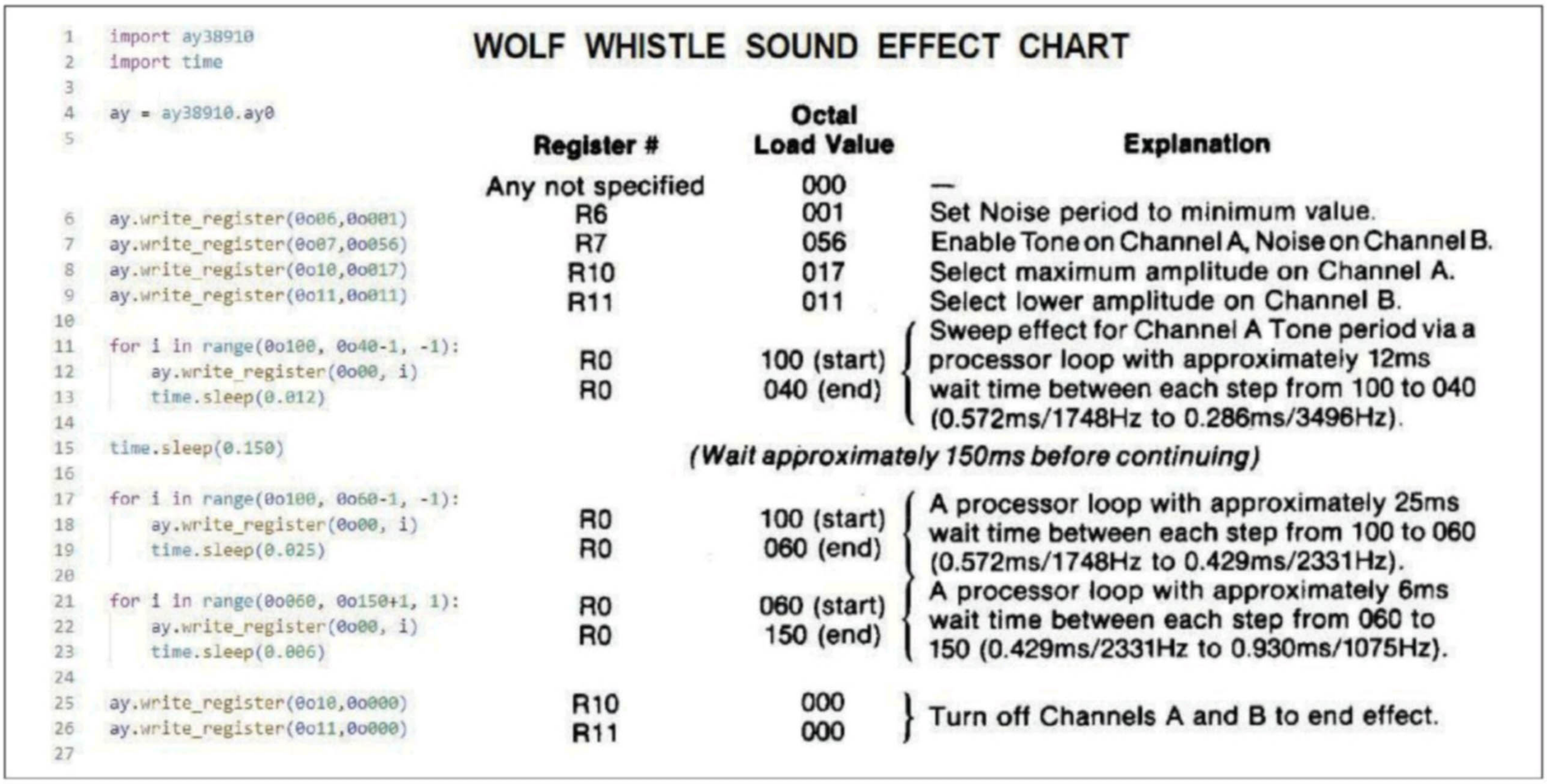
The chip is TTL compatible and understands the 3.3V logic signals correctly. You can write to it with no problems. But when you read from it, the AY-3-8910 drives the databus with 5V logic. NOTE: This will damage your Pi! My project is write-only. You will need level shifters on the databus if you want to read from the chip.

The AY-3-8910 requires a single external clock reference to generate the tones, noise, and volume envelope. All the examples in the datasheet assume a 1.789MHz clock (half the NTSC color burst frequency) [2]. Many arcade games like Frogger followed suit. The Frogger schematic [5] shows a 14.813MHz crystal, divided by eight with a 74LS393 binary counter. I used a 14.813MHz canned oscillator [6] and the same binary counter [7].

You can wire the three output voices together, and connect them directly to a powered speaker, but it sounds better with some RC filtering in the audio path. I used the filter circuit from the Arduino MIDI Chiptune Synthesizer project [8], a 3.5mm audio breakout board, [9] and a USB-powered speaker [10].

It seemed a shame to ignore all the AY-3-8910's GPIO pins. I connected a green LED to bit 0 of port A. After all my years of hobby projects, I still enjoy blinking an LED with code.

I used an Adafruit MacroPad [11] to trigger the Frogger sounds. The Raspberry Pi sees the MacroPad as a USB keyboard. Each of the 12 buttons has a neopixel backlight.



```
import ay38910
import time

ay = ay38910.ay0

ay.write_register(0o06,0o001)
ay.write_register(0o07,0o056)
ay.write_register(0o10,0o017)
ay.write_register(0o11,0o011)

for i in range(0o100, 0o40-1, -1):
    ay.write_register(0o00, i)
    time.sleep(0.012)

time.sleep(0.150)

for i in range(0o100, 0o60-1, -1):
    ay.write_register(0o00, i)
    time.sleep(0.025)

for i in range(0o060, 0o150+1, 1):
    ay.write_register(0o00, i)
    time.sleep(0.006)

ay.write_register(0o10,0o000)
ay.write_register(0o11,0o000)
```

WOLF WHISTLE SOUND EFFECT CHART

Register #	Octal Load Value	Explanation
Any not specified	000	—
R6	001	Set Noise period to minimum value.
R7	056	Enable Tone on Channel A, Noise on Channel B.
R10	017	Select maximum amplitude on Channel A.
R11	011	Select lower amplitude on Channel B.
R0 R0	100 (start) 040 (end)	Sweep effect for Channel A Tone period via a processor loop with approximately 12ms wait time between each step from 100 to 040 (0.572ms/1748Hz to 0.286ms/3496Hz).
		(Wait approximately 150ms before continuing)
R0 R0	100 (start) 060 (end)	A processor loop with approximately 25ms wait time between each step from 100 to 060 (0.572ms/1748Hz to 0.429ms/2331Hz).
R0 R0	060 (start) 150 (end)	A processor loop with approximately 6ms wait time between each step from 060 to 150 (0.429ms/2331Hz to 0.930ms/1075Hz).
R10 R11	000 000	Turn off Channels A and B to end effect.

FIGURE 3
The AY-3-8910 data manual includes the algorithms to produce several example sound effects. The Wolf Whistle chart on the right in this figure describes three code loops to sweep tone A up and down. My Python implementation of the chart is on the left. Note the register values in the code are given in octal.

You can configure each button's color and the stream of characters it sends when pressed. The MacroPad has a 128x64 OLED display to show the function of each button. You can select from several menus by turning the MacroPad's rotary encoder.

I configured four menus on the MacroPad. The "Sound Effects" menu triggers "Coin," "Squash," "Drown," "Hop," and other sounds. The other three menus select the songs from the game. Frogger is rich with music.

SOUND EFFECTS IN PYTHON

The AY-3-8910 is a simple state machine with four states of the 8-bit databus: IDLE, LATCH, READ, and WRITE. The datasheet [2] shows the values of the three control lines for each state. **Listing 1** shows my Python code to bit-bang the control lines and databus, to write a value to one of the chip's 16 registers. The inline code comments show the values of the control lines and the chip's state at each step.

You can easily translate this write_register function to your favorite language on your favorite microprocessor. The comments at the top of the listing show the same function in the Z80 code of Frogger. Despite all the arcade board's hardware glue, it is still a two-step process involving two I/O ports.

Armed with the simple write_register function, you are ready for hours of fun experiments with the AY-3-8910. Try generating tones and noises. Follow along with the examples in the datasheet. Experiment with manual volume control. Then, take the automatic envelope generator for a test drive. If you use the Python REPL, you can twiddle the AY's registers on the fly by typing Python commands.

Try writing a loop to sweep a tone from high to low and back, repeating like a siren. Make some music by playing three different notes at the same time. Figure 23 in the AY-3-8910 data manual [2] shows the values for 96 musical notes tuned to the 1.789MHz input clock. Try playing a chord—C, E, and G. For an extra 20 points, code up the song "Mary Had a Little Lamb" (it's just three notes).

The Data Manual [2] describes several examples of sound effects (such as siren, explosion, race car) and the frequencies and loops that produce them. One effect in the manual—"wolf whistle"—puzzled me when I first saw it. Can wolves whistle? We couldn't ask the Internet in the 1980s. We had to hook up the chip and write the code to find out. You can see the algorithm from the Data Manual in **Figure 3**. My Python implementation of the wolf whistle is on the left side of the figure. I used the octal constants right out of the chart. This was my first time using an octal numeric constant in Python. It felt very retro.

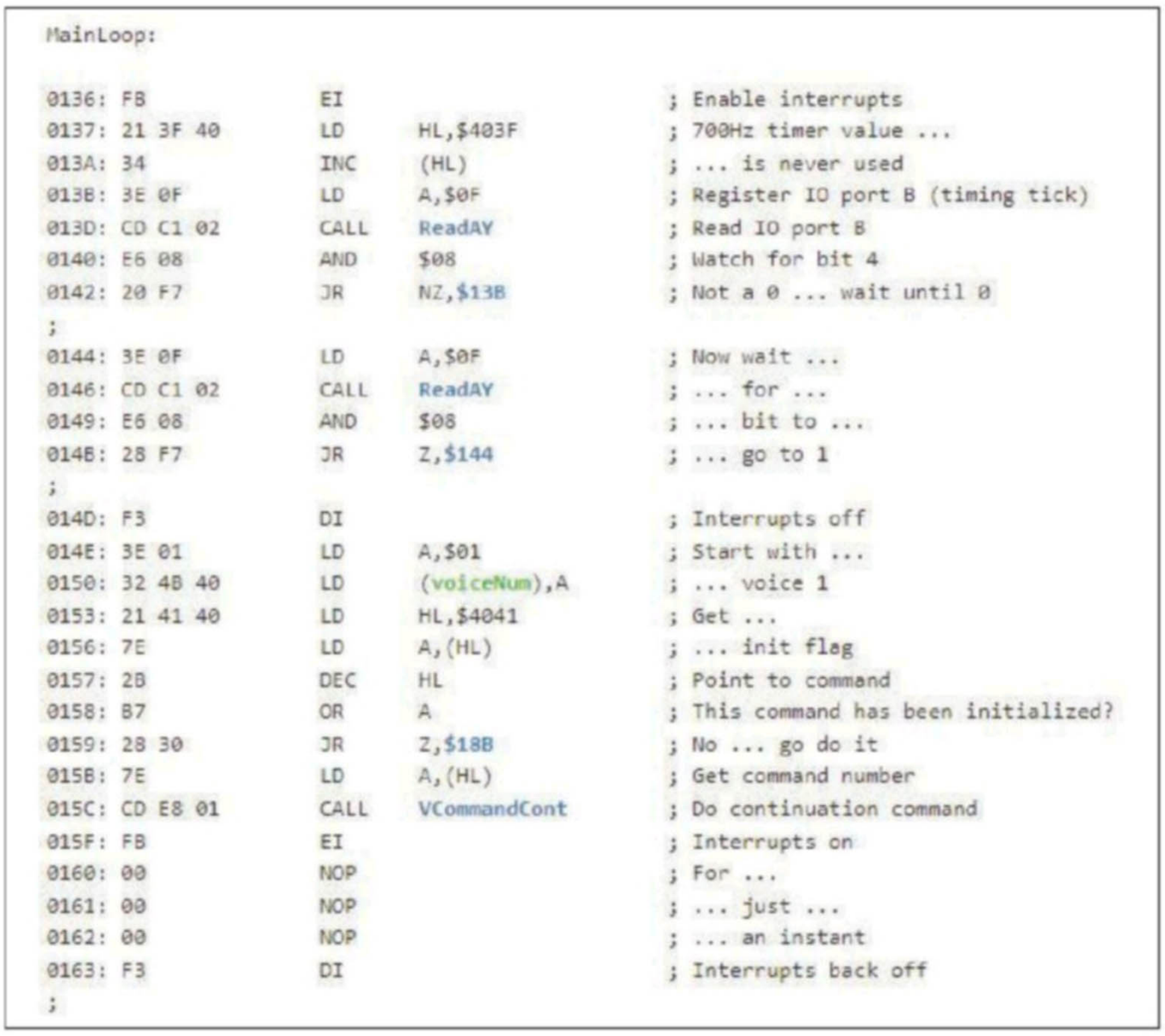
VOICE MANAGEMENT

Making simple siren and explosion sounds is one thing. But what about those complex effects we hear in the arcade? How does the game code manage multiple overlapping effects and music playing in the background? In the 1990s, I asked these questions and tracked down the binary ROM files for one of my arcade favorites, Time Pilot [12]. It uses two AY-3-8910s.

I ran the binaries through a Z80 disassembler and dug in to figure out how the code manages the AY-3-8910s. That work evolved into my Computer Archeology website, with commented disassemblies for many games, including Frogger [13].

Arcade games like Frogger and Time Pilot use dedicated sound boards with separate processors to manage sound effects. The central processor sends commands to the sound board. The sound coprocessor then manages the sound effects over time, leaving the main processor to handle gameplay.

The sound boards I looked at use the AY-3-8910 voices as a pool of available voices. When a new sound effect is requested, the code looks for an available voice and assigns it to the sound. If all the voices are busy, the code checks a table of priorities to decide which running sound effect, if any, should be preempted by the new effect. Time Pilot,



```
MainLoop:

0136: FB          EI                      ; Enable interrupts
0137: 21 3F 40    LD      HL,$403F        ; 700Hz timer value ...
013A: 34          INC     (HL)            ; ... is never used
013B: 3E 0F       LD      A,$0F           ; Register IO port B (timing tick)
013D: CD C1 02    CALL    ReadAY          ; Read IO port B
0140: E6 08       AND     $08             ; Watch for bit 4
0142: 20 F7       JR      NZ,$13B         ; Not a 0 ... wait until 0
;
0144: 3E 0F       LD      A,$0F           ; Now wait ...
0146: CD C1 02    CALL    ReadAY          ; ... for ...
0149: E6 08       AND     $08             ; ... bit to ...
014B: 28 F7       JR      Z,$144          ; ... go to 1
;
014D: F3          DI                      ; Interrupts off
014E: 3E 01       LD      A,$01           ; Start with ...
0150: 32 4B 40    LD      (voiceNum),A    ; ... voice 1
0153: 21 41 40    LD      HL,$4041        ; Get ...
0156: 7E          LD      A,(HL)          ; ... init flag
0157: 2B          DEC     HL              ; Point to command
0158: B7          OR      A               ; This command has been initialized?
0159: 28 30       JR      Z,$18B          ; No ... go do it
015B: 7E          LD      A,(HL)          ; Get command number
015C: CD E8 01    CALL    VCommandCont    ; Do continuation command
015F: FB          EI                      ; Interrupts on
0160: 00          NOP                     ; For ...
0161: 00          NOP                     ; ... just ...
0162: 00          NOP                     ; ... an instant
0163: F3          DI                      ; Interrupts back off
;
```

LISTING 2
This is the main loop in the Frogger sound code. The code waits for a 700Hz timing tick and then processes each of the three voices individually. Sound commands are signaled from the main processor with interrupts, but the code disables interrupts while processing each voice.

with two AY-3-8910 chips, has a pool of six voices (six simultaneous sound effects). Frogger, with just one chip, has a pool of three voices.

None of the games I looked at use the AY-3-8910's noise generator or its automatic envelope generator. That makes sense; there is only one set of these generators per chip, and two effects can't configure them simultaneously. Instead, the sound code controls the volume of each voice manually. For noise, the code rapidly changes pseudo-random frequency values on the voice. Different sound effects can produce different noise frequencies.

In the Frogger code, each sound effect is implemented as a pair of functions. When the main processor requests a sound effect, the first function is called once to initialize the assigned voice and data structures. After that, the second function is called at regular intervals to change the sound over time. This second function determines when the effect is done and removes itself from the scheduler.

The main loop of Frogger makes one pass through all three voices at 700Hz. Time Pilot has twice as many voices and loops at half that, 350Hz.

Listing 2 shows the first part of the main loop in the Frogger sound code. The first block of code waits for bit 3 of the AY-3-8910's port B to become 0. The second block of code waits for that bit to return high. Bit 3 is wired to the 700Hz timer.

The third block of code in Listing 2 runs the initialize or continue function for Voice 1. There are two more blocks of code (not shown) in the main loop that look exactly like this, except for Voice 2 and Voice 3. Then, the code jumps back to address 0136 to repeat the loop.

The main processor delivers sound commands to port A of the AY-3-8910. When it does, it triggers an interrupt on the sound board. The sound board's interrupt handler reads the command and looks for a free voice—or frees one up if the priority is high enough. The service routine is the sound effect scheduler. It sets the voice pointers to the desired function, and the main loop calls the voice functions one by one.

Note the "EI" and "DI" opcodes in the Z80 code. These instructions turn interrupts

```
class FroggerSound(z80.Z80Machine):
    def __init__(self):

        super().__init__()
        self.ay = ay38910.ay0

        with open(‘SoundCode.bin’, ‘rb’) as f:
            self.set_memory_block(0, f.read())

        self.set_output_callback(self.handle_OUT)
        self.set_input_callback(self.handle_IN)

        # This is start of the processing tick. This is where we trigger
        # the interrupt if there are commands to be processed. We also
        # use this as a timing point for the 700MHz timer.
        self.set_breakpoint(0x014D)

        # This is the return from interrupt command. We loop back to
        # repeat the interrupt for any queued sound commands.
        self.set_breakpoint(0x70)

        # Start the thread to read the macropad key inputs
        threading.Thread(target=self.task_read_macropad).start()

...

m = FroggerSound()
m.pc = 0x0000  # Startup goes to address 0x0000

while True:
    events = m.run()
    # Run until we hit an unhandled breakpoint
    if events & m._BREAKPOINT_HIT:
        if not m.handle_breakpoint():
            break
```

LISTING 3
The main Python code is straightforward. The constructor loads the Frogger sound code and registers my strategic breakpoints. Then, the main loop calls the "run" method to execute the emulator at native C++ speed.

on and off as needed. The main loop turns interrupts off while a voice function runs. This prevents the scheduler from clobbering the pointers and data structures while a voice function uses them. Between voices and during the long, 700Hz wait, interrupts are turned on, and the scheduler can run. Yes, young whippersnappers, we dealt with threading and exclusion way back in the old days!

Z80 EMULATION

Ivan Kosarev in the UK created a fantastic Z80 emulator written in C++ with Python bindings. You can install the native core and bindings onto the Raspberry Pi with a single pip-install command [14].

The emulator is a library compiled to fast native code. In your code (C++ or Python), you configure the core Z80 machine object. You initialize memory with your binary Z80 program. Then, you register callback functions that the core object uses for things like I/O port access.

You can also register breakpoints in the Z80 code. When the emulator's program counter hits one of these memory locations, it calls out to your code (C++ or Python) before continuing.

Finally, your main code calls the "run" method on the Z80 object, which starts the emulator running at native speeds. Occasionally, the emulator calls out to your slower Python functions. When your callback returns, the native emulator speeds off again.

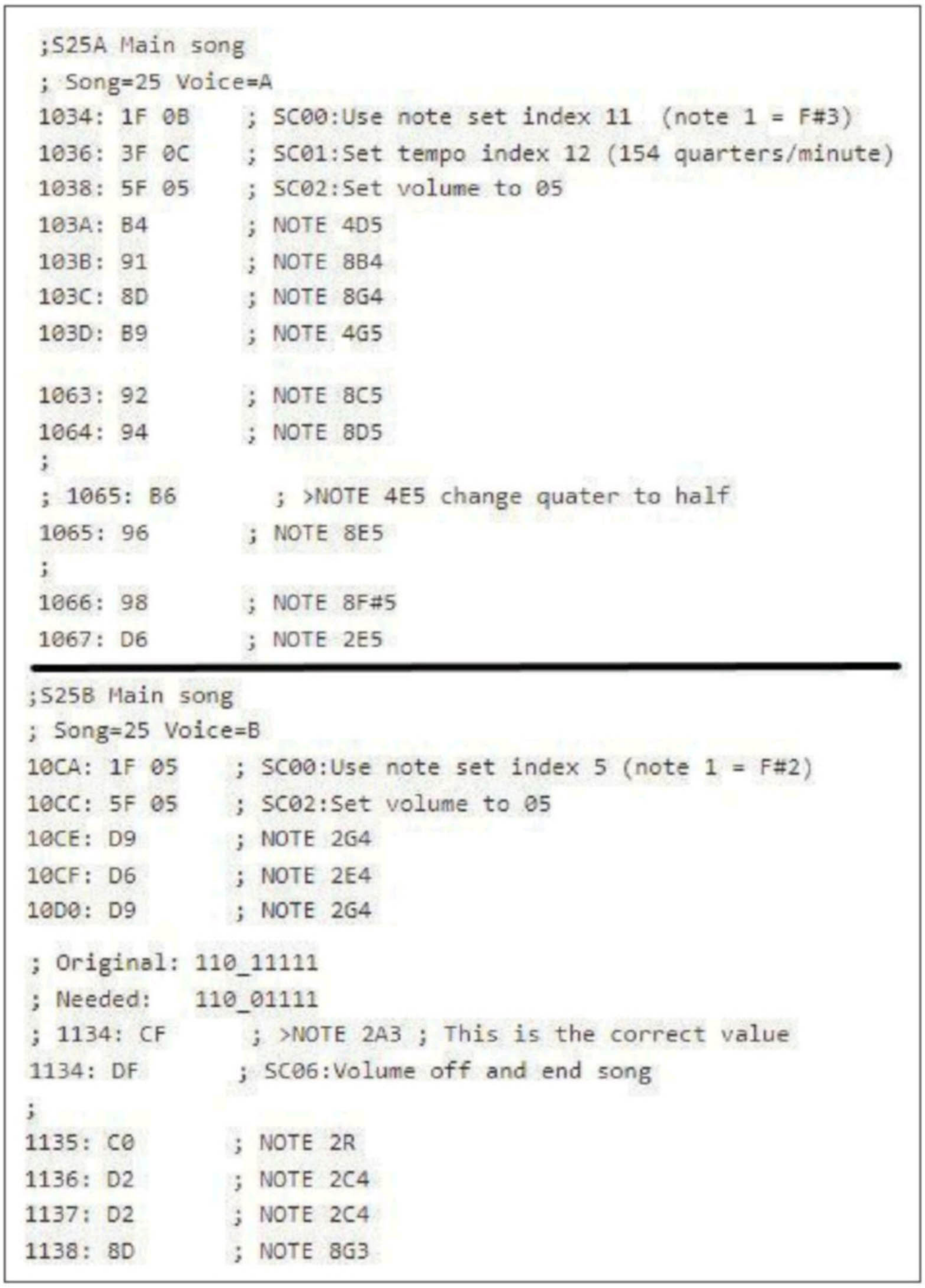
As we saw in Listing 1, the Frogger Z80 code writes to the AY-3-8910's registers through two ports: 0x40 (for the address) and 0x80 (for the value). My Python callback function passes this information to the write_ register in Listing 1.

The Frogger code reads the AY-3-8910 I/O port A (register 14) for incoming sound commands and port B (register 15) for the 700MHz timing. I used a Python list as a queue for sound requests. When you press a key on the MacroPad, the code adds the request to the end of the queue. Then, when the Z80 code reads port A, the callback pops the next value from the front of the list.

The Frogger main loop, shown in Listing 2, watches for a low-to-high transition on bit 3 of the AY port B. I handled the 700MHz timing with a breakpoint in the code I'll explain next. My callback for reading port B simply toggles the returned bit with every read. Thus, the main loop never waits for the timer bit to change—the bit is always changing.

I registered a breakpoint on address 0136, the top of the main loop. The callback for this breakpoint executes a sleep for 1.4ms (1/700Hz). This technique doesn't account for the varying time the Z80 emulator spends in the main loop, but the native emulator is very fast. Compared to the 1.4ms sleep, the Z80 code is practically instantaneous.

There are no interrupts in the emulator. Instead, my breakpoint at the top of the main loop checks the command queue. If any commands are waiting in the queue, the code changes the Z80 program counter to the interrupt service routine and returns to the emulator. The ISR pops the command from port A and schedules the new command. Another breakpoint at the end of the ISR changes the Z80 program counter back to the main loop where it came from. Thus, the interrupt is simulated in my callbacks by manipulating the program counter.



```
;S25A Main song
; Song=25 Voice=A
1034: 1F 0B      ; SC00:Use note set index 11  (note 1 = F#3)
1036: 3F 0C      ; SC01:Set tempo index 12 (154 quarters/minute)
1038: 5F 05      ; SC02:Set volume to 05
103A: B4         ; NOTE 4D5
103B: 91         ; NOTE 8B4
103C: 8D         ; NOTE 8G4
103D: B9         ; NOTE 4G5

1063: 92         ; NOTE 8C5
1064: 94         ; NOTE 8D5
;
; 1065: B6          ; >NOTE 4E5 change quater to half
1065: 96         ; NOTE 8E5
;
1066: 98         ; NOTE 8F#5
1067: D6         ; NOTE 2E5
```

```
;S25B Main song
; Song=25 Voice=B
10CA: 1F 05     ; SC00:Use note set index 5 (note 1 = F#2)
10CC: 5F 05     ; SC02:Set volume to 05
10CE: D9        ; NOTE 2G4
10CF: D6        ; NOTE 2E4
10D0: D9        ; NOTE 2G4

; Original: 110_11111
; Needed:   110_01111
; 1134: CF        ; >NOTE 2A3 ; This is the correct value
1134: DF        ; SC06:Volume off and end song
;
1135: C0        ; NOTE 2R
1136: D2        ; NOTE 2C4
1137: D2        ; NOTE 2C4
1138: 8D        ; NOTE 8G3
```

LISTING 4
This is the music script for the main song in Frogger. The note in voice A at address 1065 should be a quarter note instead of an eighth note. In voice B, the note at 1134 has a single bit flipped. Voice B shuts off at this point, even though there is more music data.

Listing 3 shows the key parts of my Python emulator code. The machine constructor reads the Z80 program from a file. This is the binary straight from the Frogger ROMs. Next, the constructor registers the port-access callbacks for IN and OUT. The constructor registers the two breakpoints in the code for timing and interrupt simulation. Finally, the code starts a thread to read keys from the MacroPad keyboard.

The main code builds the machine object, which calls this constructor. Finally, the code calls the native "run" function.

Now I'm the Frogger DJ with my soundboard. First, I push the button to play the coin-drop sound. Then, I play the intro song to start the game. I follow that up with the main song playing in the background. While that plays, I hit the "hop" button a few times, followed by the "get run over by a car" sound. Then I play the "next life" song and the main song again. It is extremely satisfying to know the original code from the arcade is generating the sounds on the original AY-3-8910 hardware.

MUSIC FIX

Frogger's musical score includes 24 multi-voice songs. You can hear all the Frogger songs and sound effects as MP3 files on my website [15]. The introduction song played when you start a new game is from a Japanese cartoon, Inu no omawari-san (The Dog Policeman) [16]. The main song is from the cartoon Rascal the Raccoon [17].

All the music is encoded as data in the Frogger sound ROM. Each note is coded as a single byte. The upper 3 bits are the note's length as a power of 2. The binary value "111" is the longest delay. I have commented this in the disassembly as a whole note. The value "110" is half as long as that—a half note—and so on. There are no ties or dots.

The lower 5 bits are the note's value, 1 through 30, with 0 being a rest and 31 indicating a special command (tempo, volume, and so on) encoded in the upper 3 bits. The note number is added to a base offset into the master note table. Thus, some harmonic voices are played in lower octaves.

When I decoded the music, I uncovered the complete main song played during the game. As a player, you could never hear all of it because the timer for the level runs out. I also discovered two errors in the music representation. You can see these mistakes and their corrections in **Listing 4**.

There is a bit flipped in a note value in the second voice of the main song. The given note value of "11111" indicates a special command, but the command number is invalid. The second voice ends prematurely when it reaches this bad note. There is more music after the bad note, but nobody has ever heard it.

I listened to the song on YouTube [18], and found the note should be a half-note A in octave 3. That's a note value of "01111". When I corrected this value in the ROM, the second voice played to completion, but the first and second voices were out of sync. When I listened to the original song again, I heard a missing eighth note beat in the main voice. The note at address 1065 in Listing 4 should be a half note, rather than a quarter note, to ensure proper synchronization in the overall composition.

I added another button option to the MacroPad to poke the corrected 2 bytes into memory. You not only can play Frogger online [19], but also can play the "original song" as coded in the ROM [20], or you can play the "repaired song" [21]. Now, for the first time, you can hear the corrected Frogger main song played in its entirety!

WHAT'S NEXT?

What's next for this project? More AY-3-8910 voices, of course. Time Pilot uses two sound chips, and I can use three more GPIOs from the Raspberry Pi for the control lines to the second chip. The two chips can share the databus.

The Frogger and Time Pilot hardware boards have quad bilateral switches (4066) to configure filtering capacitors onto each voice. There are three options for each voice: no caps, 0.047µF, or 0.22µF. I will add these switches and capacitors to my project.

How about you? Do you have a favorite arcade or computer game that used the AY-3-8910? I challenge you to make your own soundboard project with original hardware and an emulator running on the Raspberry Pi. Let me know if you do; I'd love to see and hear your work.

ABOUT THE AUTHOR

Chris Cantrell (tophercantrell@gmail.com) is an engineer for Vertiv. He also teaches for Professional and Continuing Studies at the University of Alabama in Huntsville. When he isn't working, you'll find him soldering on some fun IoT project or digging around in the ROMs of an old arcade game. Chris has written several articles for Circuit Cellar over the years.

Additional materials from the author are available at:
www.circuitcellar.com/article-materials
References [1] to [21] as marked in the article can be found there.

RESOURCES

Adafruit | www.adafruit.com

embeddedworld
Exhibition&Conference

CONNECTING THE
EMBEDDED COMMUNITY
11.–13.3.2025
NÜRNBERG, GERMANY

Get your
free ticket now!
embedded-world.de/code
Use the voucher code **GGew25**

Medienpartner

elektroniknet.de
Markt&Technik
Elektronik
Elektronik automotive
Elektronik medical
connect professional

NÜRNBERG MESSE

Basic Pulse Circuits

Part 3: Metastability, Setting, or Resetting

By
Wolfgang Matthes

In digital design, signal processing is crucial when dealing with asynchronous signals from sensors and level translators. In Part 1 of a three-part series, Wolfgang described how pulse circuits help digital circuits deal with pulse trains. In Part 2, he wrote about flip-flops, timing parameters and synchronization. In Part 3, he looks at how to mitigate metastability and ensure internal clock synchronization.

Many digital systems must process signals from outside with sensors and level translators. The signals are mostly pulse trains or bursts of pulses. Some pulses may be too narrow, some may be too wide, and other pulse trains may be affected by glitches. Designers need to learn about capturing pulses, detecting edges, filtering out or detecting glitches, and synchronizing the pulses with internal clocks. In the first part of this three-part article in *Circuit Cellar* Issue #413, we introduced the latches and flip-flops as the most basic components to address these fundamental design problems. In Issue #414, I examined how to implement the flip-flop behavior like enabling and resetting, timing parameters, and synchronization. In Part 3, I will look further into the problem of metastability and related subtleties of setting or resetting.

These are minor but intricate problems that most textbooks do not adequately cover. Solving such complex problems requires some familiarity with the appropriate components and careful attention to the minute details involved in the process.

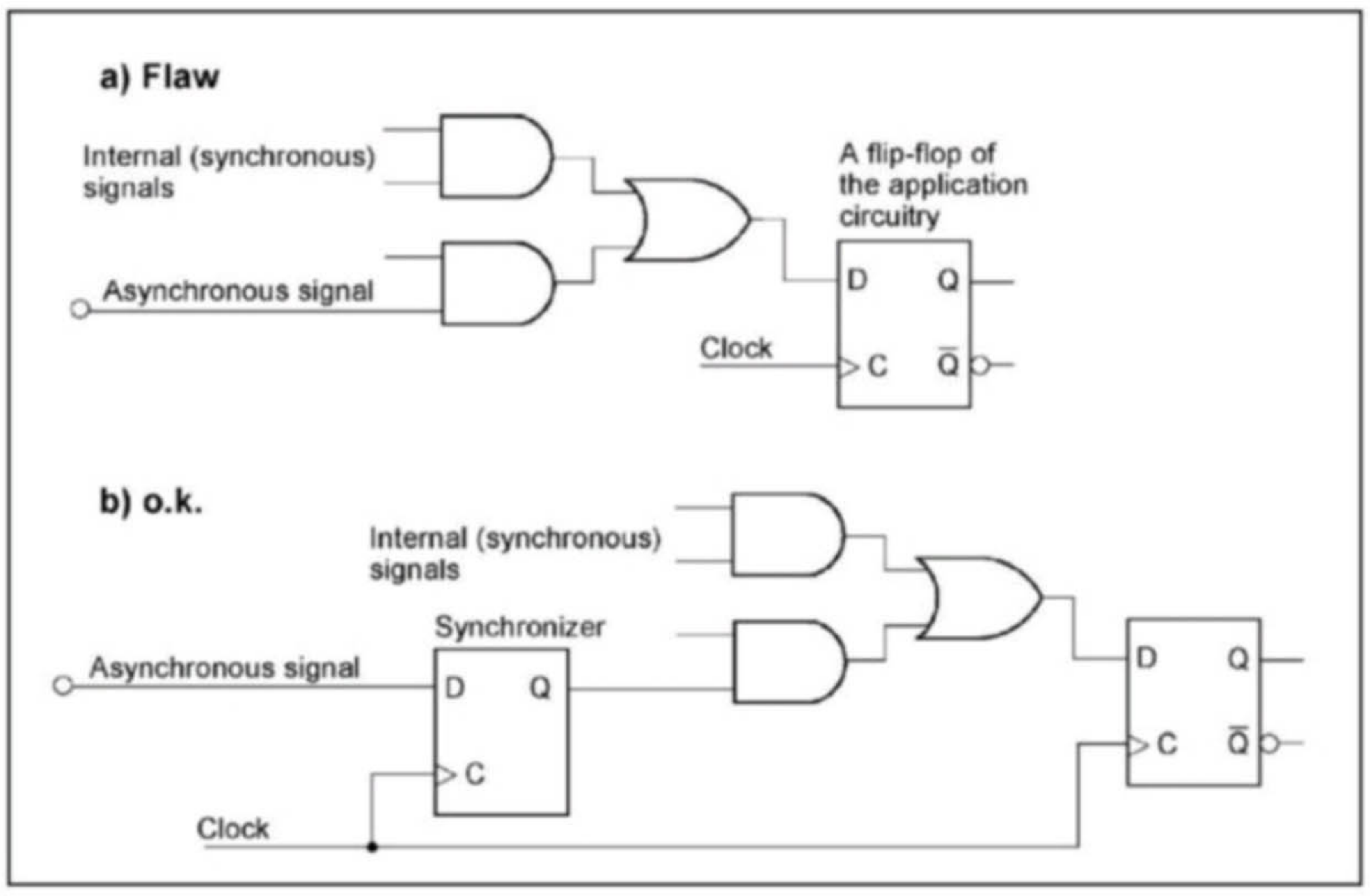

a) Flaw
Internal (synchronous) signals
Asynchronous signal
A flip-flop of the application circuitry
D
Q
Clock
C
Q̄
b) o.k.
Internal (synchronous) signals
Synchronizer
Asynchronous signal
D
Q
C
Clock
D
Q
C
Q̄

FIGURE 44
Synchronize each asynchronous signal.

METASTABILITY IN PRACTICE

Metastability problems are a typical cause of intermittent errors. During bring-up, you may bet on a metastability problem if the malfunction repeats again and again in intervals of nearly the same length. If the interval is in the size of seconds up to a few minutes, a requirement for synchronization may have been overlooked (so that asynchronous signals trickle through to the processing circuits). If the interval is larger, the designer has probably heeded the problem. Typical causes could be subtle design errors, ill-suited or faulty components, coupled-in noise, the power supply, inadequate cooling, and so on. An especially troublesome design error is to have overlooked special, rarely occurring cases in which a signal assumed switching synchronously behaves not so. In the past, I have committed one myself, resulting in a delay of two weeks because one intermittent error each hour in an otherwise perfectly working system was not easy to debug. Hopefully, up-to-date timing analysis software will find such troublesome spots early in the design process.

TYPICAL FLAWS

It is essential to synchronize all signals from outside with separate flip-flops and to provide enough settling time between synchronization and the clocks of the downstream flip-flops. Synchronize all the asynchronous signals. Do not connect them unsynchronized to downstream flip-flops (**Figure 44**). Synchronize each asynchronous signal with only one synchronizer (**Figure 45**). Otherwise, SYNC_1 might show a different signal level than SYNC_2. If you need a higher fan-out, add driver stages. When crossing a clock domain, that is, when signals generated in a circuitry controlled by a first clock are to be input into a circuitry controlled by a

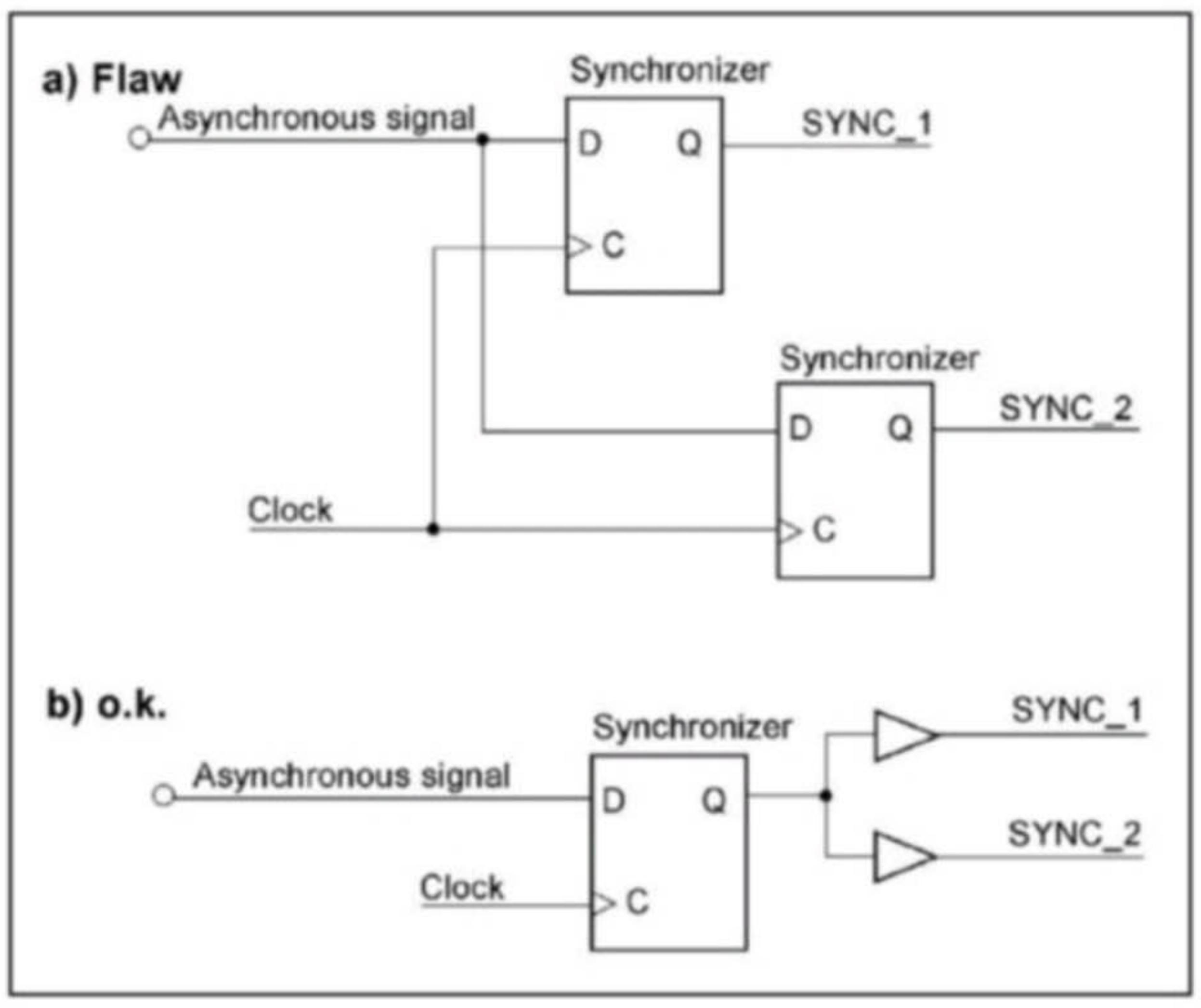

a) Flaw
Asynchronous signal
Synchronizer
D
Q
C
SYNC_1
Synchronizer
D
Q
C
SYNC_2
Clock
b) o.k.
Asynchronous signal
Synchronizer
D
Q
C
Clock
SYNC_1
SYNC_2

FIGURE 45
Synchronize each signal only once.

second different clock, do not simply lay wires from the combinatorial circuits of the source domain, but extra synchronize those outbound signals (**Figure 46**). These design tips are valid for each kind of synchronizer. To make them easy to understand, however, only single-stage synchronizers are shown here, each consisting of one D flip-flop.

EXAMPLE MTBF DATA

Truly relevant data you must obtain from the IC's manufacturer. Here, we can only show a few examples in **Table 1** and **Figure 47** and **Figure 48**.

When looking at such diagrams, observe the frequencies they are drawn for. Our examples illustrate the worst-case assumption of the input frequency being half the clock frequency. When sampling a symmetric square wave, however, we must sample the whole pulse period at least four times, so we could typically get by with a fIN = 0.25 fCLK. Thus, our MTBF expectations would become a little bit more optimistic.

Figure 47 and Figure 48 are merely illustrations (and logarithmic to boot), no precise nomograms. Therefore, calculate the quantities you need by yourself. Here you should pay attention to the desired accuracy. Do we need approximately exact values or is a rough estimate sufficient? Because we deal with exponential functions, the results may vary greatly. For example, you may get an MTBF of 70 or 200 years, depending on whether you have used somewhat more accurate or very coarsely rounded numbers.

Which MTBF should we strive for? In the highest leagues of circuit design reliability requirements are stipulated meticulously (think of aviation, space, nuclear, and the like). In our much more down-to-earth projects, we will be mostly on our own. In application notes, Internet forums, blogs, and textbooks, various design goals are mentioned, some more plausible, some less. An MTBF of ten times the average lifetime of the device or gadget could be a reasonable requirement, yielding approximately 50 to 100 years. Perhaps we could even get by with the assumption that our device will be powered off daily. Then, 10^5 to 10^7 seconds would be by far enough. Warranty periods or our own expected remaining lifetime could be further plausible objectives. Over and beyond that we could design for survival in a 1,000-year empire (think of the ancient Egyptians and Romans). Some design and calculation examples reckon even with the assumed lifetime of the sun (examples of such musings in [49]). One thousand years MTBF is no guaranteed life insurance. Despite your efforts, you may easily encounter an embarrassing failure when demonstrating your device to Gaius Julius Caesar (or, perhaps even worse, Caligula or Nero). In a nutshell, it is all statistics, which is, as a mathematically based science, not that easy. Typically, the designers leave it over to the expert people in the quality control department or to appropriate analysis software. Good starting points to become familiar with FPGA timing analysis are the manuals of the IDEs [41][51][52] and the Internet blogs and pages of appropriate user groups [42][43][44][45][46][47][48][49][56][57][58][59][60].

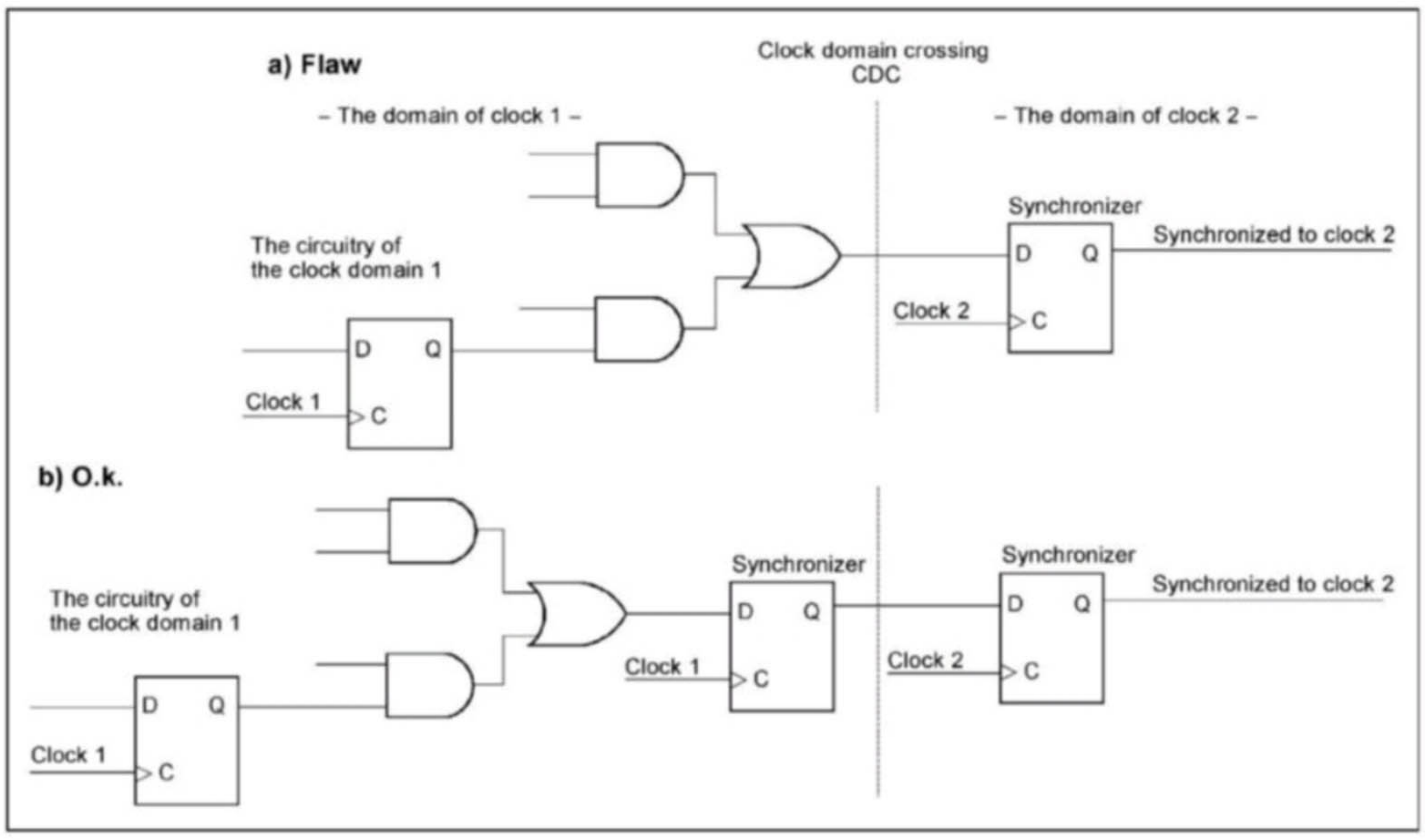

a) Flaw
Clock domain crossing
CDC
– The domain of clock 1 –
– The domain of clock 2 –
The circuitry of the clock domain 1
D
Q
Clock 1
C
Synchronizer
D
Q
Clock 2
C
Synchronized to clock 2
b) O.k.
The circuitry of the clock domain 1
D
Q
Clock 1
C
Synchronizer
D
Q
Clock 1
C
Synchronizer
D
Q
Clock 2
C
Synchronized to clock 2

FIGURE 46
Don't let output signals of combinational circuits leave a clock domain. Instead, synchronize it with the own domain's clock before entering the downstream clock domain.

Logic family	S (1/ns)	S (1/s)	T (s)
LS	74	0.74×10^9	4.8×10^{-3}
HC	55	0.55×10^9	1.46×10^{-6}
AC	28	2.8×10^9	1.1×10^{-4}
MAX	5023	5.023×10^9	2.98×10^{-17}
FLEX	12680	1.268×10^{10}	1.01×10^{-13}

TABLE 1
A few examples of MTBF data. LS, HC, and AC are SSI technologies still valid today; MAX and FLEX are somewhat seasoned FPGA series.

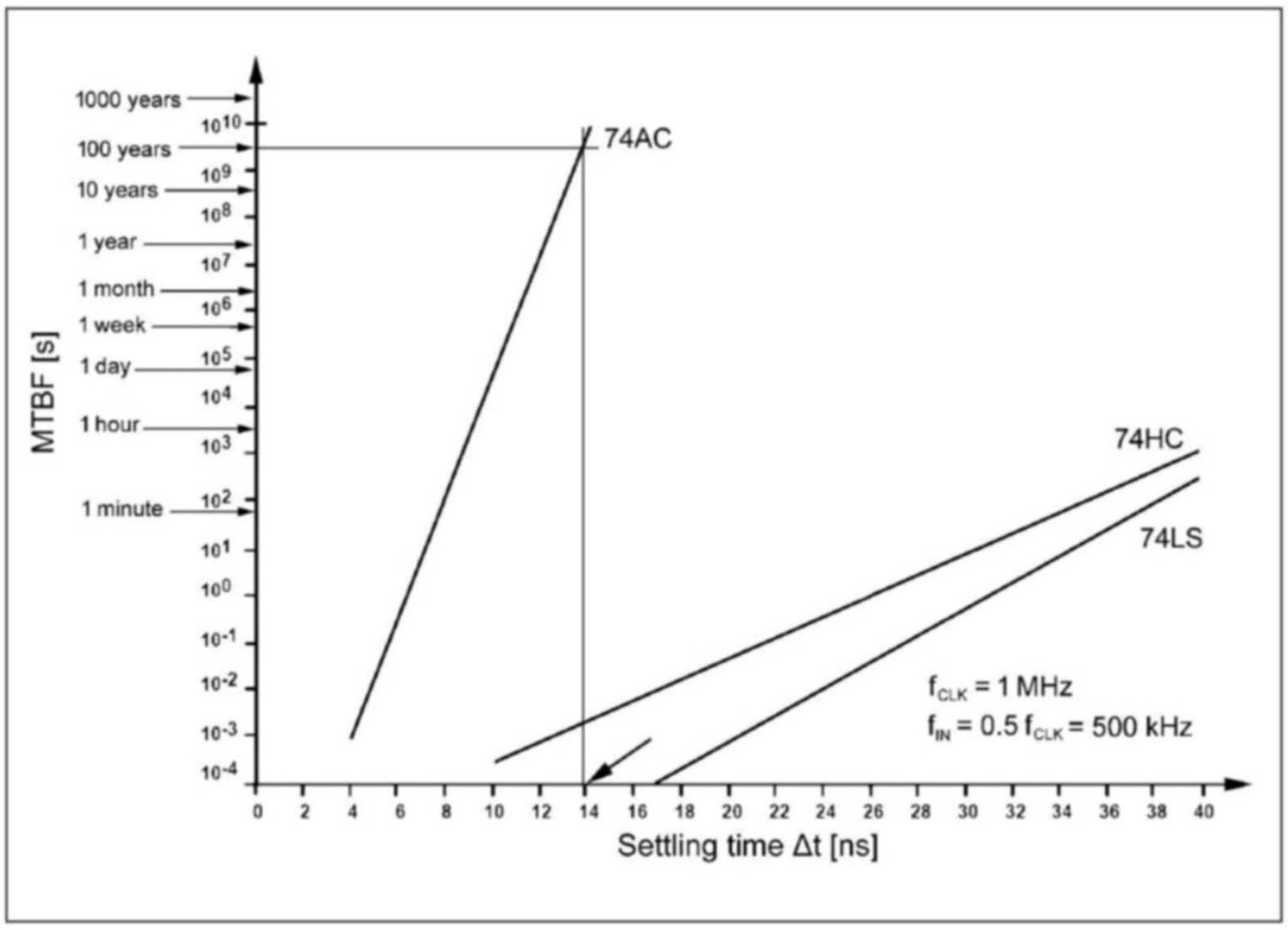


FIGURE 47

The metastable behavior of some legacy logic families. For the venerable HC and LS families, the diagram does not cover reasonable MTBF values (years). They must be calculated. So it is well-advised to follow the old rule of thumb and to select the fastest flip-flops available. Here, the AC family provides for 100 years MTBF when allowing 14 ns settling time (arrow).

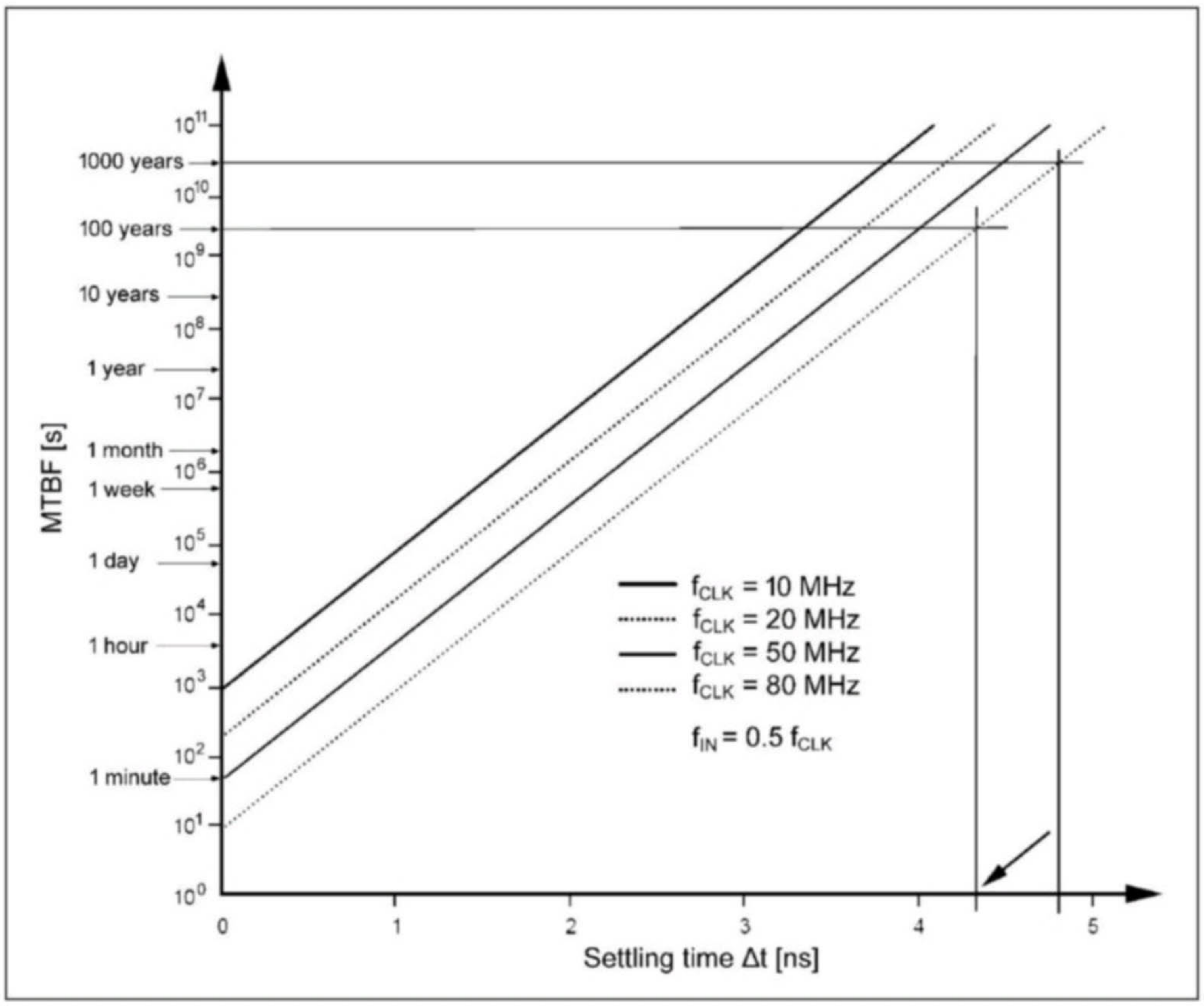


FIGURE 48

The metastable behavior of tried and testet CPLD and FPGA families. Hundred (arrow) and even thousand years MTBF can be achieved by allowing a settling time of only 5ns. Because the MTBF depends exponentially on Δt, as expressed in (3), a minuscule increase in the settling time will yield a large improvement in the MTBF. For (3) see Figure 41 in part 2.

METASTABILITY CALCULATIONS

Calculations are done best by expressing all time parameters in seconds or 1/s, and frequencies in Hz or 1/s, respectively. One day has 3600 × 24 = 86.400s. Say, approx. 10^5 s, thus leaving plenty of safety margin. One year has 3600 × 24 × 365s = 31.536.000 or approx. 3 × 10^7s.

The reciprocal of the MTBF is often referred to as a probability. This is, however, only a colloquial term. It is well-suited to grasp the problem vividly as shown here in **Figure 49**, but it is not exact. Probabilities relate to events. A probability is a dimensionless quantity between 0 (the event never occurs) and 1 (the event's occurrence is guaranteed). Strictly speaking, the reciprocal of the MTBF is a failure rate. Its dimension is 1/unit of time, for example, 1/s. With those failure rates, we can reckon as usual in the reliability theory. If our system contains n synchronizers, the overall MTBF is calculated as the reciprocal of all failure rates added together. If all synchronizers are equal and driven by the same clock, the total MTBF results of the synchronizer's MTBF divided by n.

SYNCHRONIZING THE ASYNCHRONOUS

The signals to be synchronized come from outside. Not related to any clocks, they are completely asynchronous. It is the most straightforward design task. Just attach the signals to synchronizing flip-flops or synchronizer chains.

CLOCK DOMAIN CROSSING

Clock Domain Crossing (CDC) occurs when signals come from circuits operated by different clocks. In FPGA or ASIC design, it is the most frequent synchronization problem. Think of a design comprising a PCIe hub, an USB4 host adapter, and a DDR5 memory interface controller. Such design tasks we cannot discuss here. Most designers will solve them by instantiating readily available IP cores and connecting them via standardized on-chip interfaces. When the clock frequencies are considerably lower, and the timing requirements are not that stringent (as in most embedded systems) we could consider signals out of other clock domains as completely asynchronous.

SYNCHRONIZING MULTIPLE SIGNALS

The problem concerns asynchronous signals as well as clock domain crossing. When, for example, the signals of a data bus are sampled by independent synchronizers, the synchronized outputs may differ from the arriving bit pattern. When signal and clock edges coincide, it's a matter of chance. Some flip-flops may capture the new level, while others still memorize the previous one.

There are different remedies. We could, for example, implement source-synchronous buffering in FIFOs or dual-port RAMs together with handshaking protocols that request the delivery of the data and acknowledge the receipt. A more radical approach would be bit-serial transmission via multiple independent lanes that could be synchronized individually (similar to PCI Express, only more straightforward). In small systems with relaxed timing requirements, we could synchronize the signals separately, regard differences as glitches, and treat them as such.

Figure 49 illustrates how asynchronous signals are synchronized to be read via a conventional microprocessor bus. Think, for example, of the venerable ISA bus (used even today in PC/104 systems). Some microcontroller families support similar bus systems as an alternative operating mode of their I/O ports. In this mode, the internal synchronizers are circumvented. So, the bus is directly connected to the processor core. When accessing memories or peripheral ICs, the bus cycles are already synchronized to the processor clock. This is not the case, however, when reading signals from outside. The purpose of our straightforward synchronizer is only to protect the processor core from metastable signal levels. The problem of receiving correct bit patterns must be solved by software somewhat akin to deglitching (for example, read the input several times and accept it only if it has not changed).

DISCRETE LOGIC

With the advent of integrated circuits and multi-MHz clock frequencies (we speak of the 1960s), metastability problems took the developers by surprise. Subsequent research led to formulas, parameters, diagrams, and design tips the developers could work with. When designing with tiny logic, such legacy sources are still valid, at least as a starting point. As a rule of thumb, the faster the flip-flop, the better. As you may obtain from Figure 47, Advanced CMOS (AC) yields an MTBF between 10^9 and 10^{10} s or approximately 30 to 300 years when a settling time of about 14 ns is chosen. (The calculation according to (3) yields 1.923.007.602 s or approximately 60 years.)Often, a single synchronizer stage will suffice. If not, two stages will surely do.

SETTLING TIME AND CLOCK

In most practical circuits, the synchronizer's clock will be the clock of the circuitry downstream, in other words, of the whole clock domain. The output of the last synchronizer stage is connected to the application circuitry of the clock domain (that is, the circuits implementing the functions proper). Then, the settling time Δt is a part of the total cycle time tC, as shown in **Figure 50**. It is the clock period minus the synchronizer's propagation delay tPD, the setup time tSU of the downstream flip-flops, and an application-specific delay tAD, resulting from various delays and allowances (the propagation delay of combinational circuitry, the flight time, the clock skew and so on).

Based on these premises, MTBF calculations are not that difficult. Mostly, the clock frequency and hence the clock period will be given. Then check, how much settling time is left over when all other time components have been subtracted and whether it is sufficient. In most cases, it should be more than enough. For example, imagine a clock frequency of 1 MHz (as in Figure 47) and half of the clock period available for settling. For Advanced CMOS (AC) logic, the calculator delivers an MTBF of more than 8 $\times 10^{201}$ years. So, don't worry.

CPLDS AND FPGAS

From the beginning, the flip-flops in the CPLDs and FPGAs have been designed with the metastability problem in mind. At first, with increasing speed and decreasing supply voltages, the devices improved considerably in this regard. FPGA manufacturers mentioned millions of years of MTBF if allowing for a

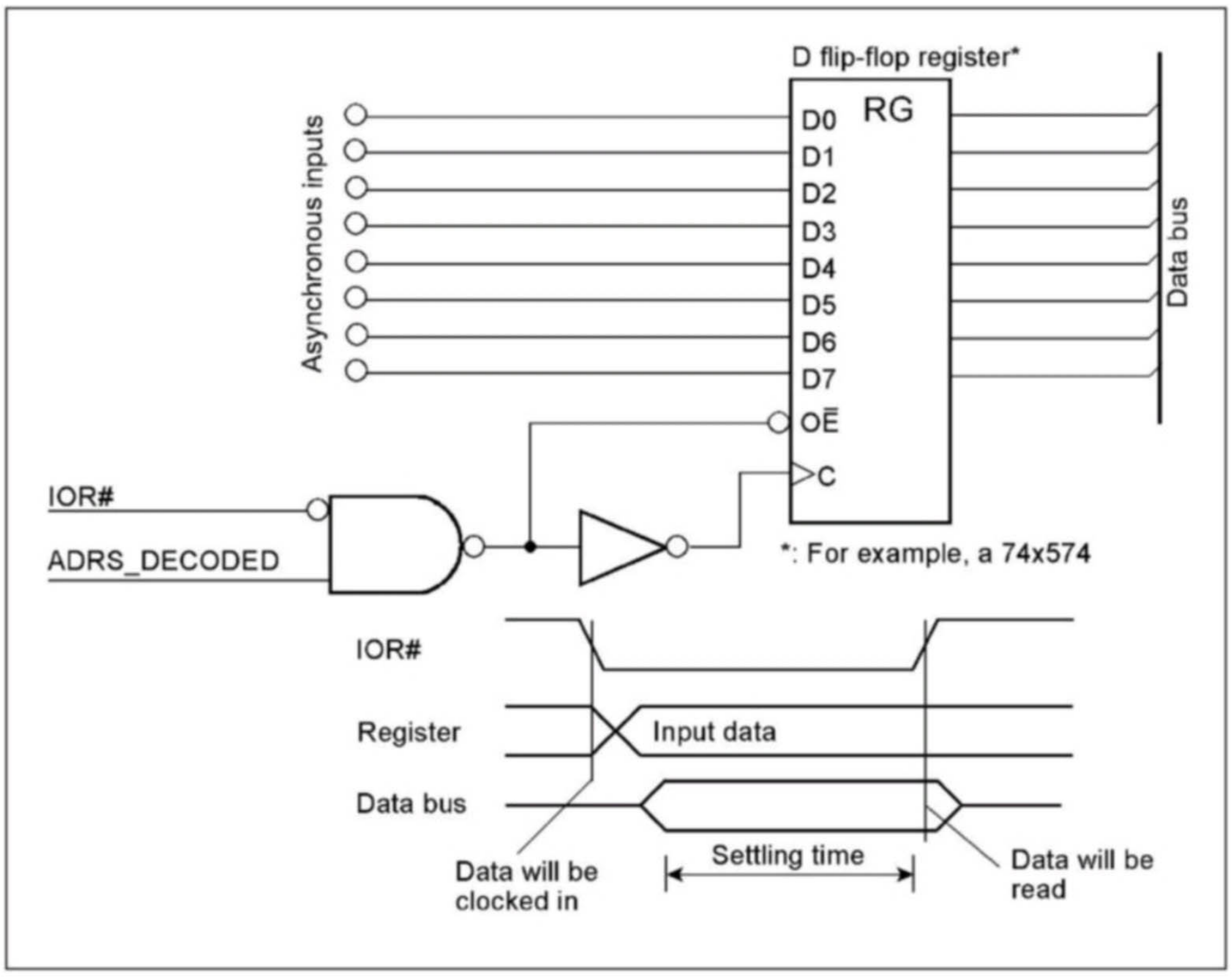


FIGURE 49

Signals to be read are synchronized to the strobe signal of a conventional microprocessor bus. The leading edge of the IOR strobe pulse clocks the signals from outside into a D flip-flop register. The register content will be read when the bus cycle ends. The width of the IOR pulse provides a generous settling time.

settling time of 5ns. Figure 48 supports the case for this magnitude too, albeit for a lower MTBF. Further shrinking geometries and supply voltages, however, exacerbate the problem. Manufacturers speak of geometries of 65nm and less and supply voltages below 1V [30]. Appropriate flip-flops show a lower closed-loop gain. Hence, they need more time to assume an acceptable output level after incurring a metastable state. Consequently, the MTBF decreases. Therefore, the manufacturers recommend at least two-stage or even three-stage synchronizers, the latter to fulfill tight MTBF requirements [40] [42]. The old saying, the faster, the better is valid here too. So, select a faster speed grade if the MTBF is not good enough, despite all design efforts.

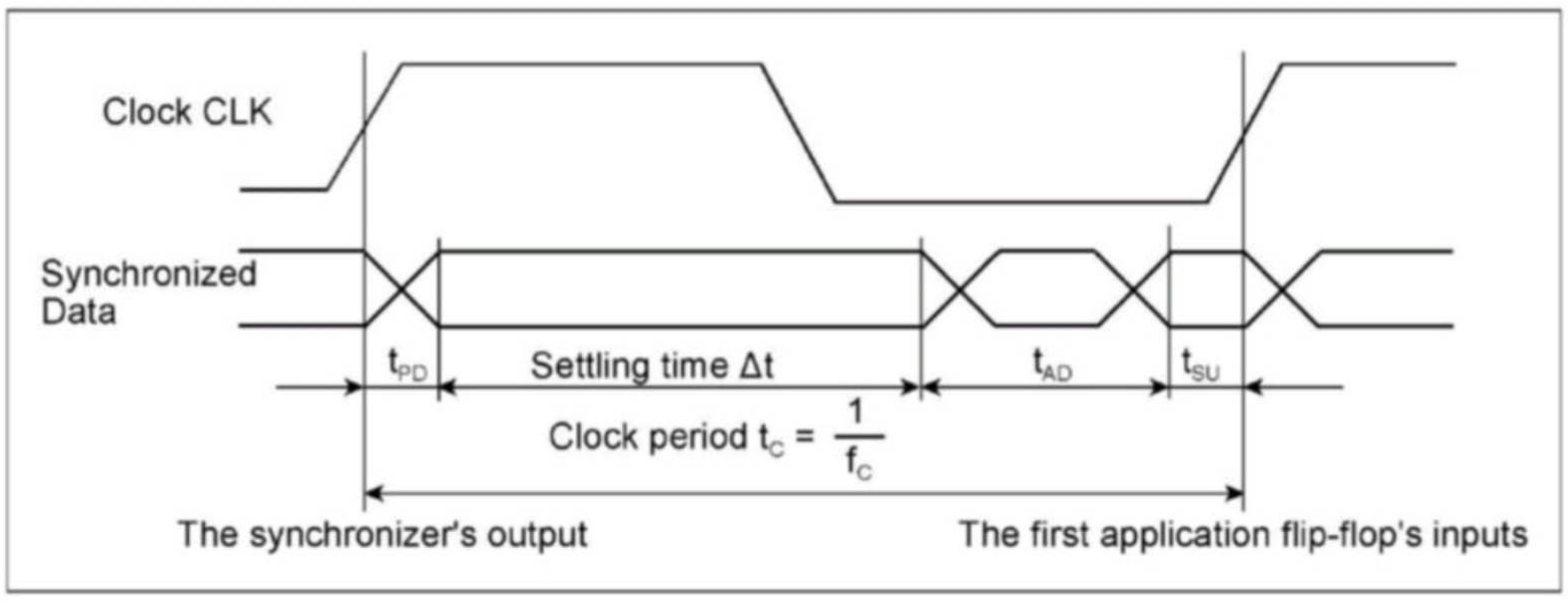

Clock CLK
Synchronized Data
t_{PD}
Settling time Δt
t_{AD}
t_{SU}
Clock period $t_C = \frac{1}{f_C}$
The synchronizer's output
The first application flip-flop's inputs

FIGURE 50
How a synchronized signal propagates from the synchronizer's output into the circuits of the clock domain.

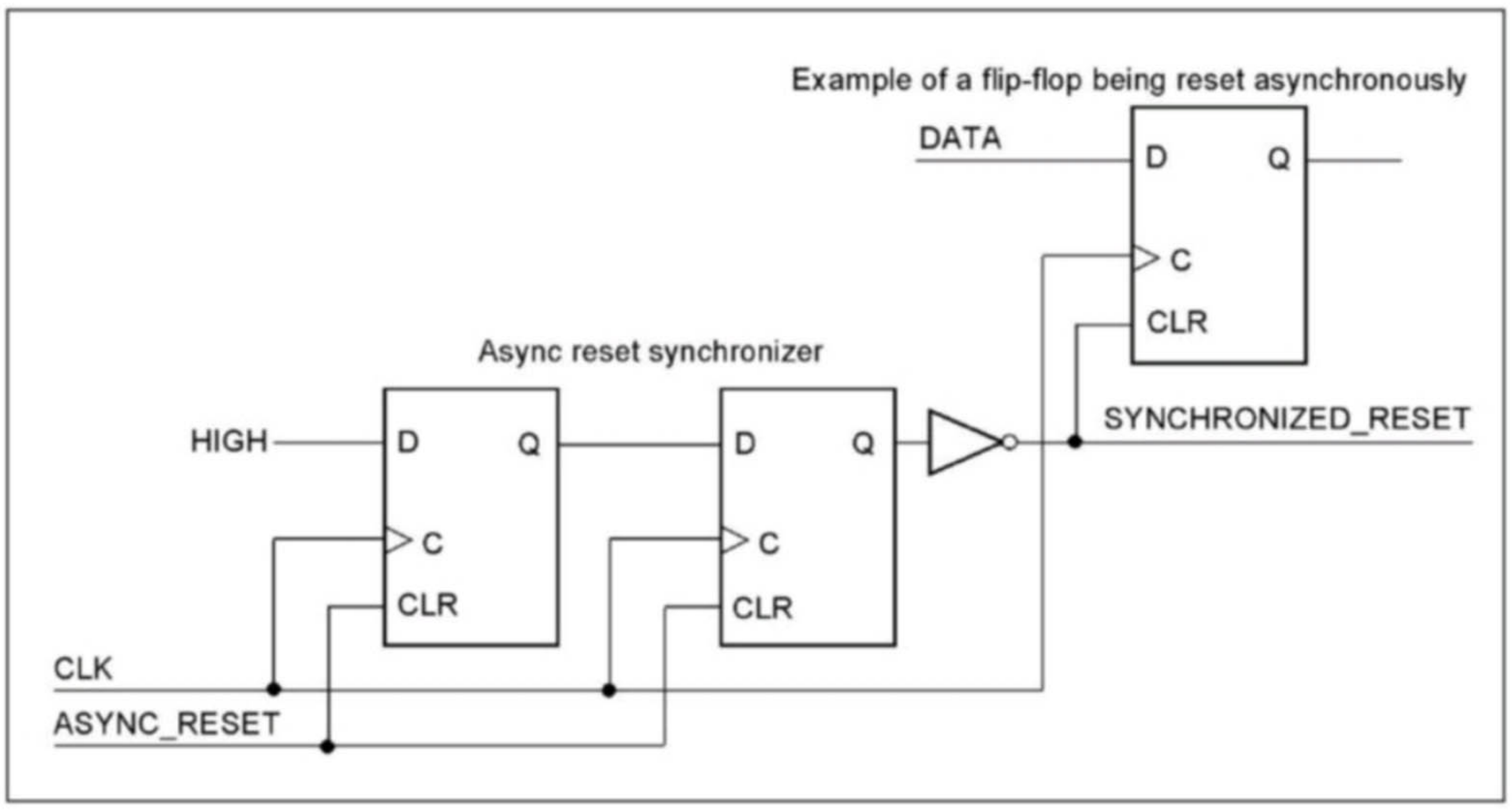


FIGURE 51
Synchronzing an asynchronous reset signal [58][59]. The resulting synchronized reset is asynchronously asserted but synchronously released. ASYNC_RESET clears all flip-flops. Because the CLR inputs are HI-active, the async reset synchronizer's output must be inverted. After being deasserted, the next two clock pulses will load the synchronizer flip-flops. Consequently, SYNCHRONIZED_RESET will be deasserted too, but synchronously.

When designing in earnest, you will have no choice but to study your manufacturer's documentation carefully and look around attentively in the appropriate Internet forums and blogs.

In contrast to Boolean optimization and state machine design, metastability is, to a considerable part, an empirical and experimental science. Therefore, be cautious that not all design software packages perform equally and that considerable differences may exist between the different FPGA series.

OVERLAPPING CLOCK AND RESET

When browsing literature of the past, you may come over numerous design ideas concerning pulse capturing, edge detection, glitch filtering, debouncing, single-shot pulse generation, and the like. Often, they make a more or less tricky use of the flip-flop's asynchronous preset and clear inputs. Implemented with SSI components, such circuits have worked for decades. The circuit manufacturers, however, generally advise against circuits that use preset and clear inputs for functional purposes. Even in the distant past, they have not endorsed it.

The reason is that asynchronous set and reset inputs of the flip-flops (preset, clear) also have critical time intervals related to the clock. Therefore, metastable states cannot be ruled out. The problem could arise if the asynchronous signal becomes inactive and a state-changing clock edge occurs during this time.

The data sheets of the flip-flops typically specify a setup or recovery time (tSU, tREC), which indicates how long the asynchronous signals (Preset, Clear) must be at least inactive before the next state-changing clock edge may occur. Otherwise, the clock pulse could be ineffective or could even cause a metastable state. Some legacy components show a benign behavior. (I remember a design seminar in the 1990s where the lecturer noted wryly that the old SN7474 is still in demand because of its suitability for such tricky applications.) For new types, however, you may never take it for granted. Therefore, use Preset and Clear only for initialization and, maybe, error recovery. When designing for CPLDs or FPGAs, you may be well advised to study the appropriate app

Additional materials from the author are available at:
www.circuitcellar.com/article-materials
References [41] to [60] as marked in the article can be found there.

RESOURCES

AMD | www.amd.com
Intel | www.intel.com
Lattice Semiconductor | www.latticesemi.com
Microchip | www.microchip.com
Nexperia | www.nexperia.com
NXP | www.nxp.com
STMicroelectronics | www.st.com
Texas Instruments | www.ti.com

notes and chapters in the architecture manuals. As an example, **Figure 51** shows how an asynchronous reset signal could be fit in such an environment. It is asserted asynchronously, so it will immediately reset the flip-flops. It is, however, released synchronously, so no timing requirements will be violated when the flip-flops are commencing their work. By the way, being careless regarding reset and initialization may lead to nasty design errors that are hard to find by simulation and during bring-up. The latter because they occur so seldom, perhaps once per day or week (and then, according to Murphy's law, first during the endurance tests preceding manufacturing release).

SUMMARY AND SUGGESTIONS

Today, digital design will be done mostly by instantiating readily designed subsystems as IP cores and by high-level behavioral descriptions. Such approaches, however, cannot cover all real-world problems. Occasionally, we have to cope with minor details such as synchronizing signals from outside, capturing pulses, suppressing glitches, and the like. To solve such design problems, we must be acquainted with the fundamental components. Here, we've take the combinational gates for granted and concentrate on latches and flip-flops, the so-called sequential components. In contrast to the combinational gates, they have a state characterized by the single bit they have memorized. To use them for detecting, sampling, synchronizing, delaying, and so on pulses, we need some understanding of intrinsic details not found in most textbooks, like the problem of metastability, the limits of the clock frequency, and the different ways of setting or resetting. Design ideas handed down from the old days will not always be helpful. What has worked well with TTL components must not work with tiny logic or inside a CPLD or FPGA. What we strive for is to keep asynchronous circuitry and tricky flip-flop circuits as straightforward and small as possible and solve the remaining design tasks by synchronous circuits in a CPLD or FPGA.

ABOUT THE AUTHOR

Wolfgang Matthes has developed peripheral subsystems for mainframe computers and researched special-purpose and universal computer architectures. He has also taught MCU Design, Computer Architecture, and Electronics at the University of Applied Sciences, Dortmund, Germany. He has filed more than 50 patent applications and written 14 books. He is still busy writing books. You may reach him at: info@realcomputerarchitcture.com

Precision PIN Diodes for Superior MRI Performance
MICROCHIP
MPP4000
microchip.com/PINDiodesMRI
MICROCHIP

Engineering Healthcare

Cutting-Edge Technologies to Enhance Patient Care

By
Caleb Smith

Explore the groundbreaking advancements in Pulsed Field Ablation technology, where engineering meets cardiac care to revolutionize atrial fibrillation treatment. This article delves into the intricate design and functionality of cutting-edge devices that enhance precision and safety in medical procedures, showcasing the vital role electronics play in shaping the future of healthcare.

Medicine has always depended on the best technology of its time. From the flint tools used for trepanation of human skulls 10,000 years ago to the modern advancements in 3D printing of organs and tissues through stem cell research, it is the progress in science and technology that has saved countless lives. Today, the integration of electronic systems is an essential part of medicine at all levels, from diagnosis to treatment both in and out of the hospital. The heart itself runs off of electrical conductive signals, and the monitoring of these via EKGs has long been a common practice in patient monitoring. While this monitoring may be easily achieved through the unobtrusive placement of conductive stickers, the treatment of disruptions to this conduction is a more complicated matter. New technology known as Pulsed Field Ablation, made possible through advancements in electrical engineering, seeks to treat one of these disruptions, and has shown remarkable promise. Through improvements in catheter ablation technology, and real-time mapping technology, Pulsed Field Ablation technology is overtaking traditional methods such as thermal ablation in the treatment of atrial fibrillation, with its improved safety and efficiency.

ATRIAL FIBRILATION

Atrial fibrillation is a dysrhythmia of the atriums of the heart caused by rapid or irregular electrical activity (**Figure 1**), and is a common cause of cardiac arrest, and therefore relies on cutting-edge solutions and technologies for its treatment. Aside from years of medical therapy, the currently accepted treatment has, for decades, typically consisted of ablation: the creation of small holes in cardiac tissue to destroy tissues responsible for abnormal activity, typically around the pulmonary vein (the vein responsible for bringing oxygenated blood from the lungs to the left atrium of the heart) This treatment has been achieved primarily through a process called thermal catheter ablation, in which a thin, flexible catheter has been engineered to be inserted into the body—typically through the groin—and fed to the atrium where abnormal cardiac or pulmonary vein cells are destroyed through the precisely controlled delivery of thermal energy. The two subcategories of this: radiofrequency ablation and cryoablation use hot and cold temperatures respectively, with radiofrequency ablation being the more common of the two, accounting for approximately 40% of all ablation procedures, approximately double that of cryoablation.

In the past year, however, a new method of ablation has begun to be utilized: pulsed field ablation, a groundbreaking technique utilizing high-voltage electrical pulses to surgically remove targeted tissues with enhanced precision and reduced damage to surrounding tissues. Using high amplitude electrical pulses, this method has shown tremendous ability to destroy irregular tissue while decreasing procedure times, and reducing the chance of atrial fibrillation recurrence, yielding a more permanent solution to the problem.

BOSTON SCIENTIFIC

Boston Scientific has emerged as one of the leaders of PFA technology, and is currently one of two companies with Pulsed Field Ablation technology on the market. Their product, Farapulse, seen in **Figure 2**, currently accounts for 47% of Pulsed Field Ablation procedures. Farapulse uses a 12Fr (4mm) diameter, 115cm over the wire Farawave catheter with 20 electrodes capable of precisely and accurately delivering bipolar and biphasic waveforms from the Farastar generator. The Farapulse catheter is engineered with two distinct configurations—basket and flower—each optimized for electric field delivery and penetration during different phases of the ablation procedure. During the procedure, these configurations are each rotated once, and deliver two paired 2.5 second pulsed field applications in the ostium (opening of the left atrium to the pulmonary vein) to ensure electric field penetration. This has shown a significantly lower damage threshold when compared to conventional methods, such as thermal ablation, and decreases damage to the esophagus, and phrenic nerve, with a 17,000 patient trails showing no reported esophageal fistulas, permanent phrenic nerve palsy (paralysis) or pulmonary vein stenosis (narrowing) highlighting its selective targeting of cardiac myocytes while perservng adjacent tissue. Additionally, pulsed field ablation typically causes myocytes (cardiac muscle cells) to die from apoptosis, a more regular and controlled cell death than necrosis and with less inflammation.

With magnetic tracking, this Faraway Nav Pulsed Field Ablation Catheter is compatible with the Faraview mapping software module. The Faraway Nav Pulsed Field Ablation Catheter uses advanced data acquisition techniques to gather geometric data on its environment to be capable of creating a real-time 3D rendering of the heart in reference to the catheter for surgeons and technicians to reference during the procedure. This can eliminate the need for the insertion of an additional mapping catheter or for the exchange of catheters mid-procedure. This module uses Field Tag technology to visualize the intersection of the generated fields with the myocardium of the heart in real time by displaying their areas of effect within the heart. Additionally, the module uses field tags to track the positioning, rotation, and deployment of the catheter during the eight-field application process and later confirm the overlap of these applications, again eliminating the need for extra voltage application mapping catheters.

A multinational survey of more than 17,000 patients was conducted in the two-year window between March 2021 and March 2023 to evaluate the safety and viability of Farapulse pulsed field ablation technology after receiving FDA approval. This study collected data from approximately 91% of centers offering Farapulse pulsed field ablation treatments(106 centers in total) and 413 operators, providing critical insights into real-world clinical performance. This study aimed to determine if any damage

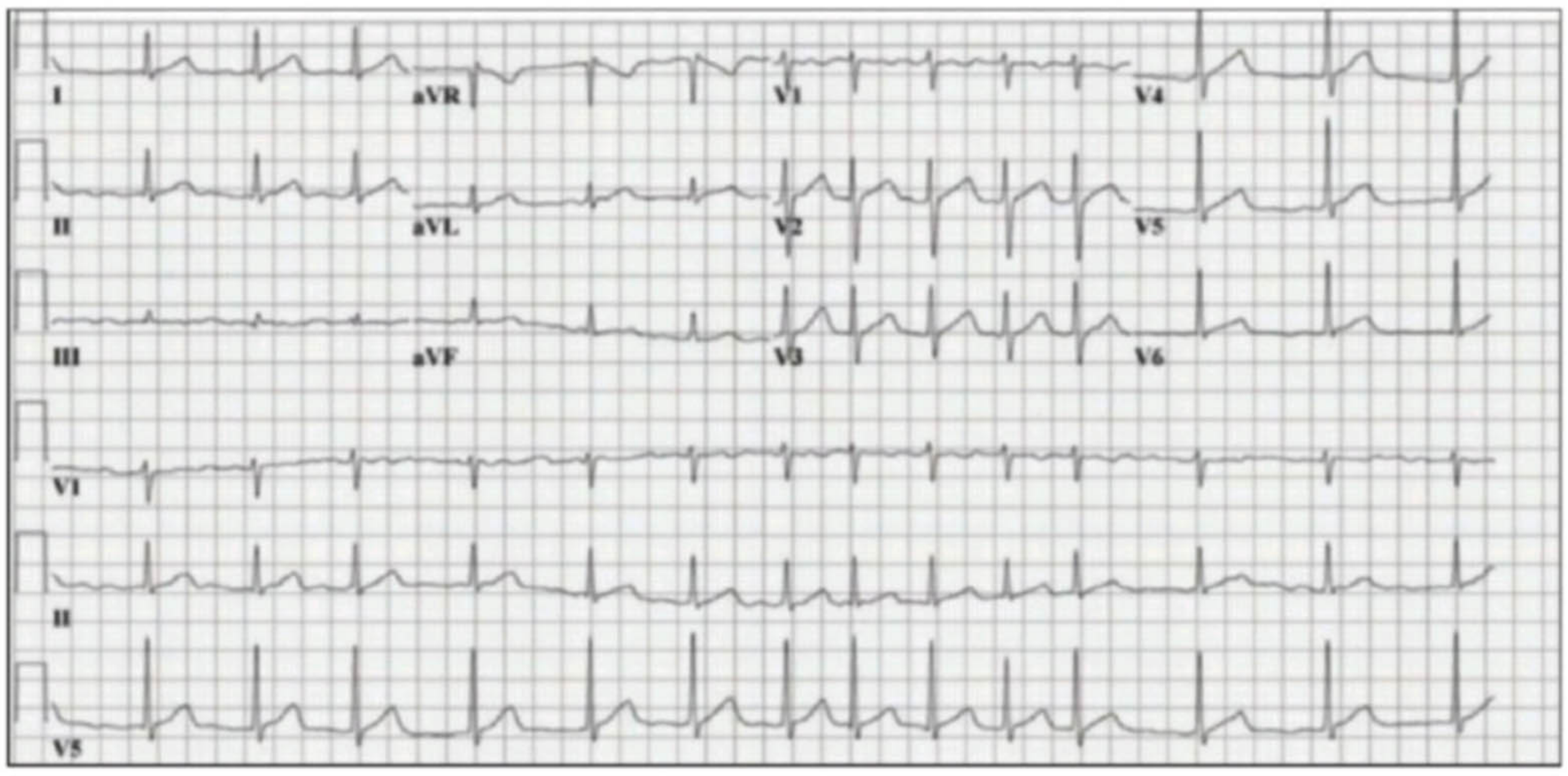


FIGURE 1
An EKG readout shows the disruptions caused by atrial fibrillation.

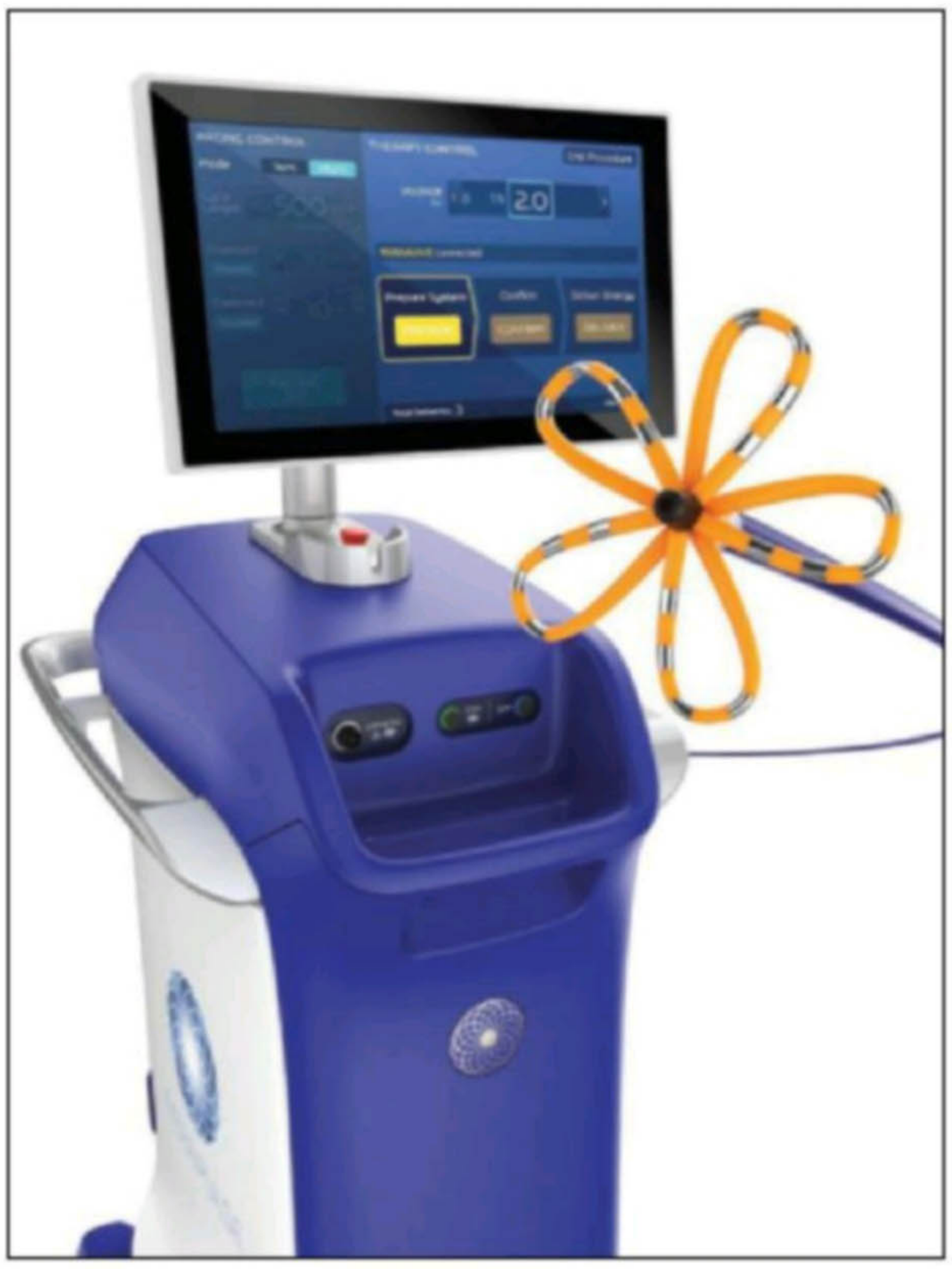


FIGURE 2
Boston Scientific's Farapulse PFA system, and its catheter. (Image source: Boston Scientific)

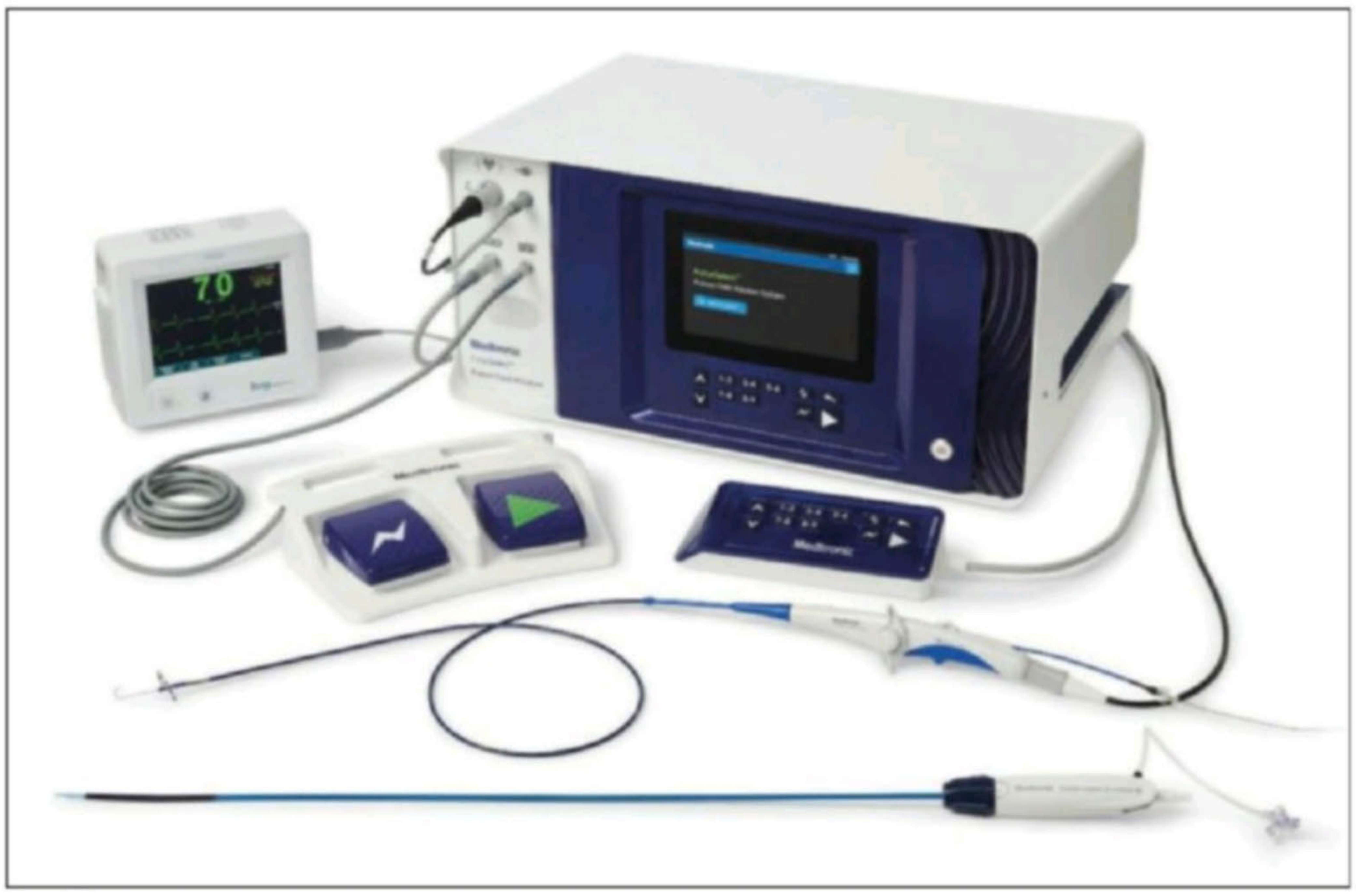


FIGURE 3
A view of Medtronic's PulseSelect system, including its generator, catheter, and remote control. (Image source: Medtronic)

to the surrounding tissue—specifically the esophagus and phrenic nerve—or unusual effects resulted from the use of Farapulse pulsed field ablation treatments. This study found a "major adverse effect" rate of 0.98%, with 36% of these being accounted for by pericardial tamponade (fluid around the heart) underscoring the safety of this procedure, and providing engineers with critical data to inform future designs and procedural protocols. This study also concluded a 0.03% mortality rate, with five deaths occurring in the 17,642 participants. The article specifies, however, that three of these deaths were not related to the ablation procedure.

MEDTRONIC

Alongside from Boston Scientific's Farapulse pulsed field ablation system, Medtronic's PulseSelect pulsed field ablation system (**Figure 3**) was one of the first to reach the market, and comprises a majority of systems currently in use. Medtronic's PulseSelect pulsed field ablation system is designed to allow for mapping of the heart and precise lesioning through biphasic waveform optimization pulsed field ablation. The system consists of the PulseSelect PFA catheter, and PulseSelect PFA generator.

The PulseSelect PFA catheter is a 9Fr (3mm) over the wire catheter with a 10Fr (~3.3mm) bidirectional braided FlexCath Contour steerable sheath with nine evenly spaced electrodes for field delivery that deploy into a 25mm diameter circular array tilted forward 20 degrees to allow for consistent contact with the tissue surrounding the pulmonary vein. This catheter runs along an included j hook guidewire, and comes equipped with the FlexCath Cross which allows for the integration of a 21-gauge transseptal needle. This technology eliminates the need for guidewire, needle, and therapeutic sheath exchanges creating a zero sheath exchange operation with only a 50 minute left ventricle dwell time during trial procedures, and only 30 seconds of energy delivery per pulse.

This catheter is powered by the PulseSelect PFA Generator: a small 250V input device capable of outputting a 1500V bipolar biphasic waveform controlled via the PulseSelect PFA remote control, capable of delivering a test pulse to scan for proximity to the phrenic nerve, and PulseSelect PFA foot switch for hands free control during operation, with built in overcurrent protection for patient safety. Additionally, the system comes with the PulseSelect PFA Electrogram cable for connecting the generator to external electrophysiology monitoring equipment. Finally, the system comes with 7600EP Cardiac Trigger Monitor, which uses a four-lead ECG to find the peak of the patient's R wave to enable synchronized pulsed field ablation delivery through a BNC-BNC interface cable to synchronize pulsed field ablation delivery with R wave peak in a process known as R wave gating.

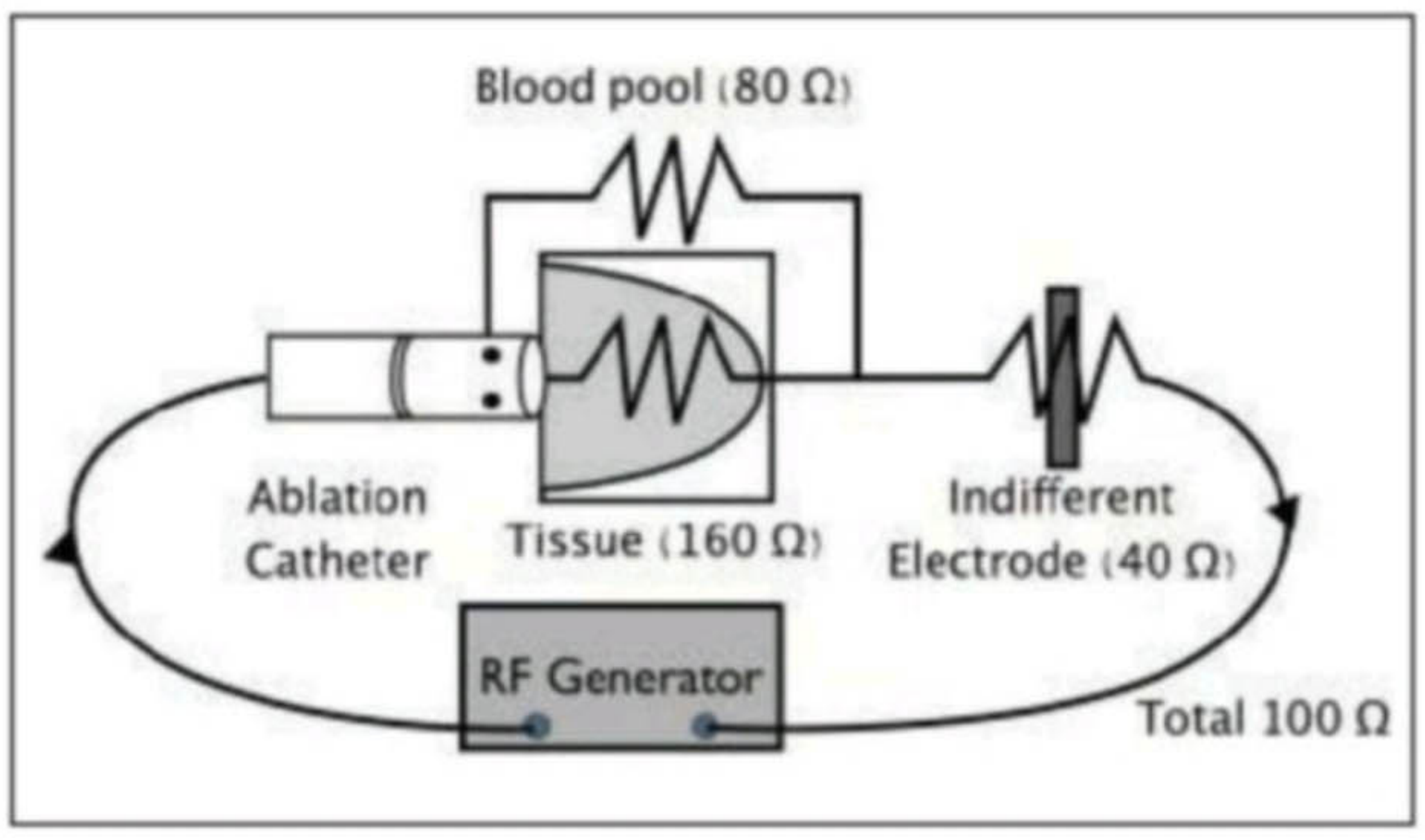


FIGURE 4
Radiofrequency ablation kills unwanted tissues through heat generated by an alternating current, run through a needle to generate ohmic heating. (Image source: Science Direct)

RADIOFREQUENCY ABLATION

Despite its rapid growth in the market, pulsed field ablation technology has not yet supplanted established techniques, such as radiofrequency thermal ablation, which continues to demonstrate versatility across a wide range of applications. Despite the benefits of pulsed field ablation discussed at the beginning of this article, the diversity of uses for radiofrequency ablation help it to remain a useful tool in medical science. Not only have RFA systems been designed to treat atrial fibrillation like FaraPulse and pulse select, but also in the ablation of tumors, venous closure in areas unable to be accessed by intrusive surgery, pain management, treatment of skin lesions and varicose veins.

In contrast to pulsed field ablation, which utilizes electroporation of tissues (increased permeability of cell membranes due to the application of an electric field) to induce targeted cellular apoptosis, radio frequency ablation kills unwanted tissues through heat generated by an alternating current, typically less than 500kHz in frequency, run through a needle inserted into selected points within the

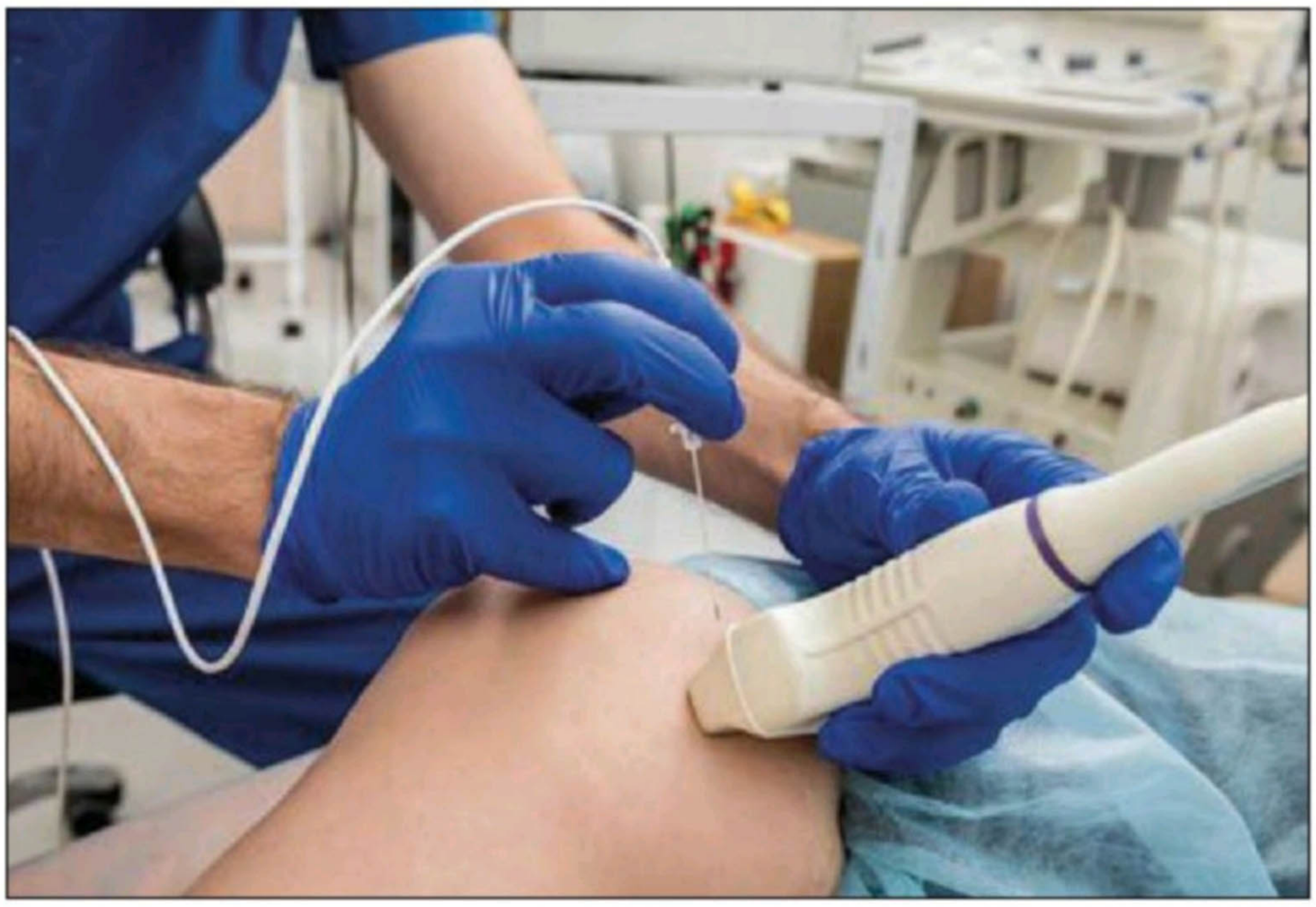


FIGURE 5
A surgeon uses ultrasound to guide a Radiofrequency Ablation catheter during a procedure.

tissue to be ablated to generate ohmic heating. The unipolar circuit (**Figure 4**) created by this electrode is completed through a ground placed on the patient's body. If the tissue to be ablated is highly conductive, however, this current will be carried to surrounding tissue and cause thermal damage to them. Additionally, the thermal conductivity of the tissues in question may carry heat beyond the distance which electricity in currents great enough to cause significant damage may be carried. When heating occurs, the tissues and proteins in question will denature and coagulate, changing their conductivity, and if charred, increase their impedance, leading to decreased current density, and decreased efficacy of lesions. For these reasons, precise control is required for not only the power of the catheter but also its temperature, with catheters typically being cooled through open irrigation, in which saline is sprayed around the catheter as ablated tissue, or closed loop irrigation, in which saline is run in a closed loop in the catheter for convective cooling.

Unlike the built-in mapping and imaging of Farapulse, most radio frequency ablation technology's guidance relies on X-ray fluoroscopy of the inserted catheter for guidance and mapping. Like traditional fluoroscopy, this relies on the image produced by X-rays on a reactive screen, on which the patient lies for the duration of the procedure. Otherwise, they rely on ultrasound guidance, as seen in (**Figure 5**), where the catheter is inserted and guidied in real time, via ultrasound imaging. While capable of producing high-resolution images in real time, this technique has not shown the same capability for generating 3D models as other techniques possible with modern science.

CRYOABLATION

Under the same umbrella category of "thermal ablation" as radio frequency ablation, but on the other end of the spectrum, cryoablation uses extreme cold (as low as -75°C) to destroy cells and ablate tissue (**Figure 6**). By freezing these tissues through a cryoprobe attached to a large cryogenic freezing unit, or sometimes through direct deposition of cryogenic liquids or gases, ice crystals form within the cells of the tissue, destroying their membranes and organelles in a process similar to frostbite. Additionally, this freezing coagulates blood in these tissues, disrupting the flow of oxygen, leading to induced apoptosis(programmed cell death.) Unlike with radiofrequency ablation, electrical conductivity of the tissue is not a concern for this form of ablation procedure.

While normally used to treat tumors, cryoablation can be used to treat cardiac dysrhythmias by causing thermal scarring of the conductive paths to reroute the electrical conduction of the heart to a more regular and beneficial pattern. While radiofrequency ablation is the more common thermal ablative procedure, cryoablation demonstrates a particular advantage in that it can be used to map tissues. By temporarily cooling the tissue to a reversible point (ex., -10°C) the outcome of ablation of this tissue can be observed,

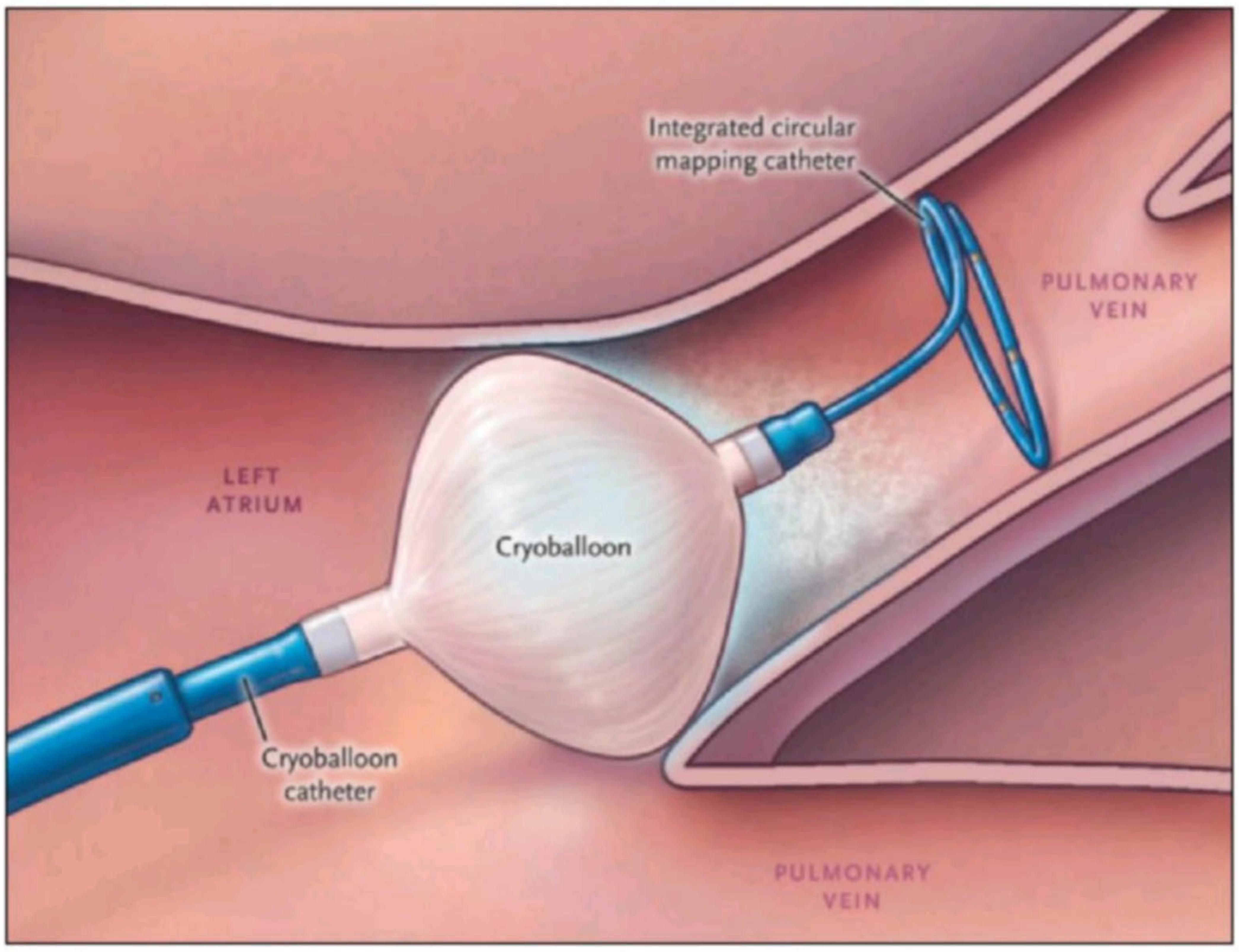


FIGURE 6
In the cryoablation system, MCUs manage the operations including temperature control, pressure monitoring, and timing of the ablation process.

FIGURE 7
ADAM series SBCs are engineered to meet stringent medical standards. (Image source: Advantech)

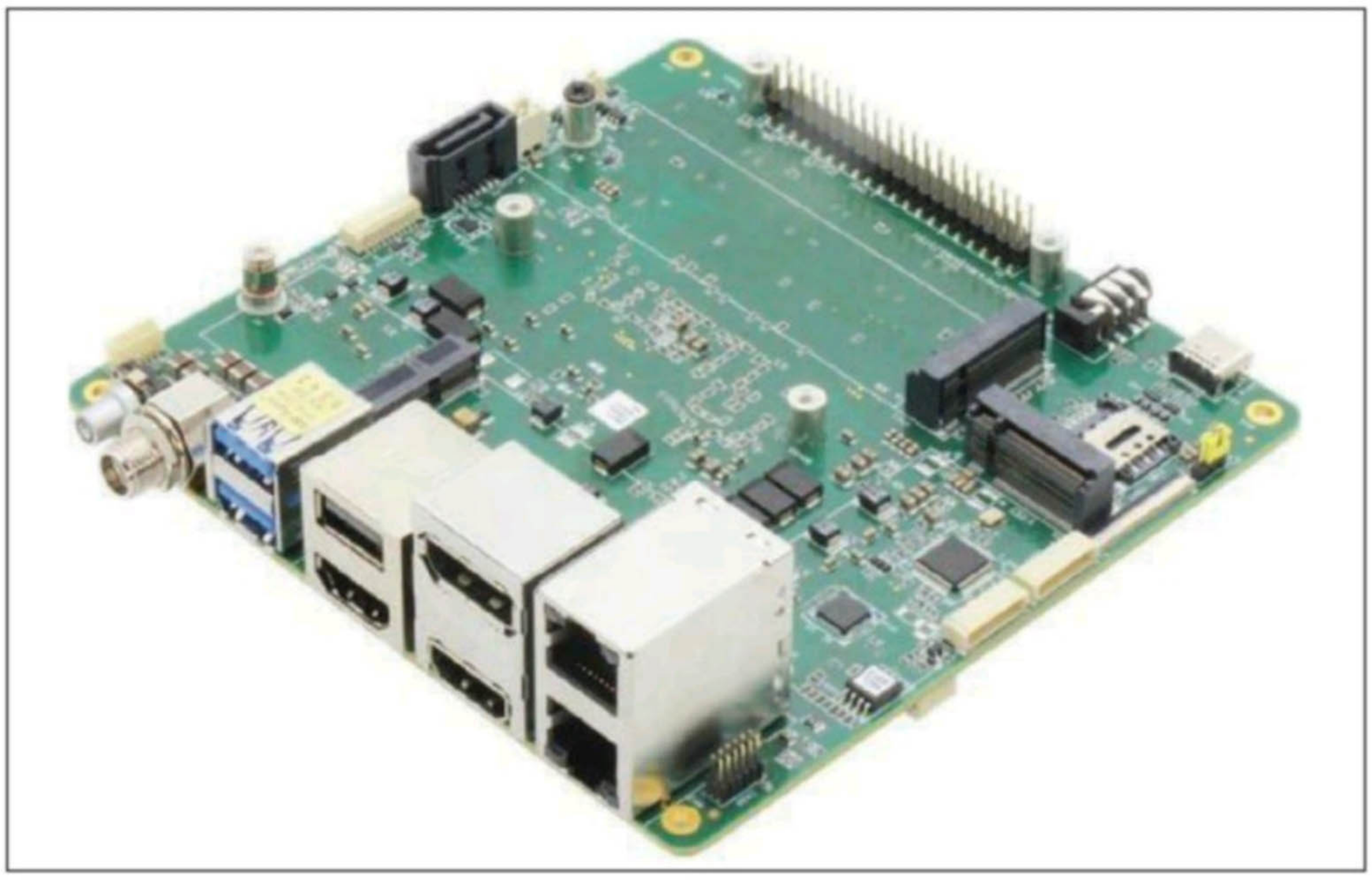


FIGURE 8
AAeon UP Xtreme i14 Developer Board. (Image source: AAeon)

and if the effects are undesirable, reversed by rewarming the tissue. This, in addition to traditional imaging technologies such as ultrasound and laparoscopic imaging, form a complete real-time picture of the body area and tissue in which are to be operated.

As discussed in this article, this new pulse field ablation technology shows significant improvements over previous ablation procedure technologies. This new technology has so far been proven to provide quicker procedure times, with minimal risk to patients and a lower rate of atrial fibrillation recurrence. More importantly however, this new technology has shown an increased ability to specifically target cardiac myocytes leaving surrounding tissue unaffected, decreasing damage to the esophagus and phrenic nerve typically resulting from thermal ablation procedures. Due to precise field generation technology, and real-time cardiac mapping, pulsed field ablation technology such as Boston Scientific's Farapulse, and Medtronic's Pulse Select have made ablation procedures for the treatment of atrial fibrillation more safe and effective than ever before.

BEHIND THE SCENES

While the exact specifications and chipsets of these ablation technologies are mostly proprietary, there there are many electronics companies that have products geared specifically for medical use. Addressing the concerns of medical applications such as power usage, size restrictions, reliability, robust processing, and compliance with medical standards.

Advantech offers a line of industrial-grade single-board computers (SBCs) specifically designed for medical applications, such as the ADAM series (**Figure 7**). These SBCs are engineered to meet stringent medical standards, providing features such as long-term availability, extended temperature ranges, and enhanced durability. With powerful processing capabilities and integrated connectivity options, Advantech's SBCs facilitate seamless data acquisition, real-time monitoring, and efficient communication in medical devices. Their modular design allows for easy customization and integration into various healthcare solutions, from patient monitoring systems to diagnostic equipment, making them an ideal choice for medical device developers seeking dependable and scalable computing solutions.

Aaeon UP Xtreme (**Figure 8**) SBCs are engineered to support a variety of medical devices, including diagnostic, patient monitoring systems, and telemedicine solutions. Featuring powerful processing capabilities, flexible I/O options, and

Additional materials from the author are available at:
www.circuitcellar.com/article-materials

Resources

Advantech | www.advantech.com

Aaeon | www.aaeon.com

Boston Scientific | www.bostonscientific.com

congatec | www.congatec.com

Medtronic | www.medtronic.com

Microchip | www.microchip.com

NXP Semiconductor | www.nxp.com

Texas Instruments | www.ti.com

integrated connectivity to facilitate real-time data processing and communication in healthcare settings. They adhere to strict regulatory standards, ensuring reliability and safety in critical applications.

Congatec has a spectrum of embedded computer technologies and AI ecosystems (**Figure 9**). For medical device engineers, there is a variety of ecosystems to allow for the best fit. The PICMG COM-HPC Computer-on-Module standard is built to fuel the development of the next-generation of connected medical computers and edge servers with massive bandwidth and connectivity.

Microchip is enabling medical device innovators to develope advanced software, optimizing industrial design, and navigating the complexities of medical regulatory compliance. An example is the SAMA7D65 Microprocessor paired with the Microchip Graphic Suite (**Figure 10**). This user-friendly, comprehensive solution streamlines the integration of graphical user interfaces (GUIs), animations, and images into touchscreen displays. It caters to the medical sector by delivering high-performance processing and sophisticated graphics rendering, crucial for modern medical devices that demand real-time data visualization and intuitive user interactions. Additionally, the MPU's low power consumption and robust security features ensure dependable and secure functionality in critical healthcare environments.

Texas Instruments provides a suite of microcontrollers and processors specifically tailored for medical applications, notably the MSP430 family of low-power microcontrollers. These devices are engineered for portable medical systems, facilitating smarter, high-performance healthcare solutions through their analog and embedded processing capabilities. Their products find extensive use in various medical applications, including patient monitoring systems, diagnostic instruments, and wearable health technologies.

The LPC5500 MCU series from NXP Semiconductors harnesses Arm's Cortex-M33 technology, offering substantial architectural enhancements and increased integration compared to previous generations. This series boasts significant reductions in power consumption and advanced security features, such as SRAM PUF-based root of trust, real-time execution from encrypted internal flash images, and asset protection through Arm TrustZone-M. Furthermore, the LPC5500 series includes seven scalable families with diverse package and memory options, complemented by the comprehensive MCUXpresso Software and Tools ecosystem and affordable development boards. This MCU series is a vital component of NXP's EdgeVerse edge computing platform, making it well-suited for modern medical applications.

COM-HPC
congatec

FIGURE 9
congatec ecosystems for medical development. (Image source: congatec)

MICROCHIP
GRAPHICS SUITE
MICROCHIP

FIGURE 10
Microchip Graphic Suite. (Image source: Microchip)

CONCLUSION

With several new pulsed field ablation technologies scheduled for launch in 2025, new technologies and improvements to existing technologies may be seen on the market very soon. As the MCU, SBCs and development systems improve the precision, accuracy, and reliability of pulsed field ablation tools also improve. All the technology development goes hand in hand to solving many other medical problems, improving traditional methods and leading to entrily new solutions.

Datasheet: System-on-Module Technology Key IoT Enabler

High-Performance Computing Systems Packed into Compact Form Factors

By
Tom Murphy,
Editor-in-Chief

System-on-Module (SoM) technology is gaining traction across various industries, driven by the demand for compact, integrated solutions that allow developers to add AI and connectivity options as demands evolve. Explore the factors fueling the SoM growth and the advantages these modules offer.

In the era when inferencing is needed precisely where information is being collected and processed, it's important to remember that the latency of the cloud is the most important aspect of Artificial Intelligence that system designers want to remove from their system performance equations. How else could we bring massive amounts of computing performance closer to the action than with a system-on-module (SoM) technology? It can deliver multiple terra operations per second (TOPS) right where the application needs it most, according to the literature supporting many of the datasheets profiled here.

On the following pages, you will see SoMs from AAeon, Axiomtek, BeagleBoard, Nvidia, Toradex, Variscite, SolidRun, and even the latest Raspberry Pi CM5.

Processors steal the spotlight of many SoMs, and here, you'll find representatives from Intel, AMD, NXP, Broadcom, and Nvidia. If you see processor vendors here that aren't usually associated with PC platforms, they were most like likely selected for efficiency and longetivity.

SoM presents the ultimate method of bringing computing power right to the edge. Its signature compact size characteristics entice developers to choose full-fledged PC/server-type compute density where only a cue-card-sized space is available.

If you need to speed up the pace of machine vision visual inspection rapidly, an SoM product on the market today will give you that capability. Suppose your automated guided vehicle (AGV) keeps delivering burgers and fries to the wrong address. In that case, it's possible to swap out the old SoM and replace it with one producing next-generation navigation capabilities to ensure the AGV gets to the right place without slamming into someone's shin bones.

The primary driver behind the increasing adoption of SOM products is the need for space-efficient and cost-effective computing solutions. SoMs integrate essential CPUs, memory, and I/O interfaces into a tiny card-sized board that is easy to remove and replace. This compact design is particularly beneficial for applications in the Internet of Things (IoT), industrial automation, and consumer electronics, where minimizing physical footprint without sacrificing performance is crucial.

Continuous improvements in semiconductor technology significantly enhance the performance and appeal of SoMs. Innovations in processing power and energy efficiency allow SoMs to meet the demands of next-generation applications that require higher computational capabilities. These advancements make SoMs attractive for developers looking to build more powerful and energy-efficient devices.

The proliferation of IoT devices and the shift toward edge computing are significant factors driving SoM adoption. As businesses seek powerful processing capabilities and reliable connectivity for real-time data processing, SoMs provide the necessary infrastructure. They enable data analysis and decision-making closer to the data source, which is critical for applications requiring immediate responses.

SoMs facilitate quicker product development cycles by simplifying hardware design. Manufacturers can customize functionalities to differentiate their products and respond swiftly to market demands. This flexibility is essential for companies that maintain a competitive edge in fast-paced environments.

In the context of Industry 4.0, SoMs are pivotal in driving automation and connectivity within smart manufacturing. They support robotics, predictive maintenance, and digital twins, which enhance productivity and operational efficiency. By integrating SoMs into manufacturing systems, companies can leverage data for improved decision-making and resource management.

NEXT MONTH'S TOPIC: DSP & DSC Send related product announcements to editor@circuitcellar.com

AAEON UP Xtreme i11

The UP Xtreme i11 is powered by Intel's 11th Gen Core processors, providing high performance for AI and edge computing applications. With support for up to 32GB of DDR4 RAM, it offers extensive I/O options, including dual HDMI outputs and USB 3.0. The board is designed for reliability and efficiency, making it suitable for demanding environments. Its compact size and flexible expansion options make it ideal for embedded systems in various applications, including smart cities and industrial automation. The UP Xtreme i11 supports a wide range of peripherals, enabling developers to create customized solutions.

- Intel 11th Gen Core (i3/i5/i7) processor
- Up to 32GB DDR4 RAM
- Up to 1TB NVMe SSD storage
- Dual HDMI 2.0 outputs
- Dual Gigabit Ethernet ports
- PCIe and M.2 expansion slots
- Compact dimensions: 90mm x 90mm
- Low power consumption
- Wide operating temperature range
- Extensive software support
- Real-time clock support
- Integrated security features
- Multiple audio interfaces
- High thermal efficiency
- Ideal for smart city applications
- Flexible power management

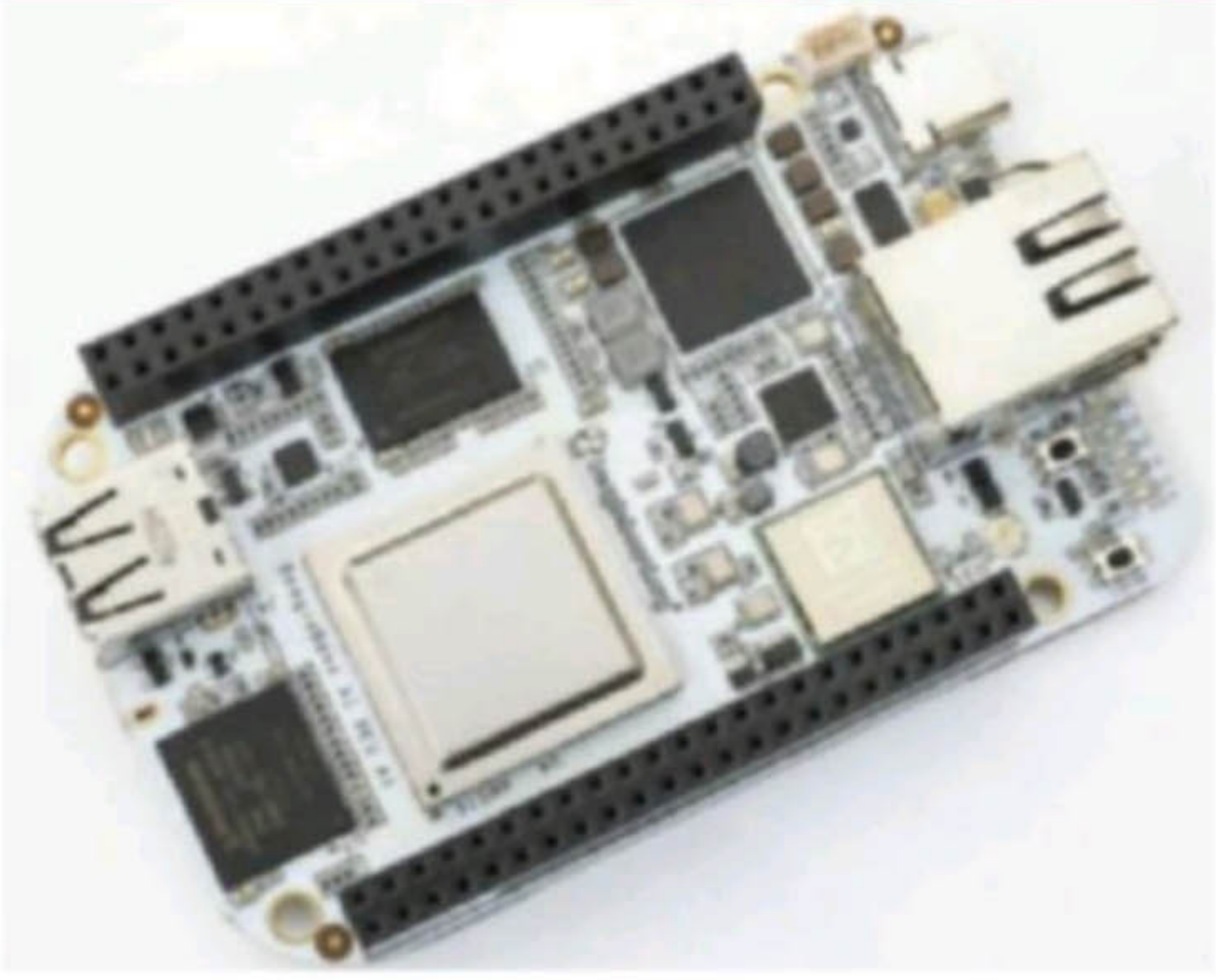
AAEON
www.aaeon.com

Axiomtek CAPA13

The Axiomtek CAPA13 is powered by AMD Ryzen Embedded V1807B/V1605B APUs, providing needed performance for industrial applications. With support for up to 32GB of DDR4 RAM, it offers a variety of I/O options, including dual HDMI outputs and USB 3.0. The board is designed for high reliability and efficiency, making it suitable for demanding environments. Its compact size and extensive connectivity make it an ideal choice for embedded systems in automation, transportation, and smart city applications. The CAPA13 also supports a range of expansion options for customization.

- AMD Ryzen Embedded V1807B/V1605B APUs
- AMD Radeon RX Vega Graphics
- Up to 32GB DDR4 RAM
- Up to 1TB SSD storage
- Dual HDMI 2.0 outputs
- DisplayPort++
- 1 x LVDS; 18/24-bit single/dual channel
- Dual Gigabit Ethernet ports
- PCIe expansion slots
- HD audio link
- Temp range: -20°C to +60°C (-4°F to 140°F)
- Compact dimensions: 146mm x 104mm
- Wide operating temperature range
- Extensive software support
- Designed for industrial applications
- Real-time clock support
- Integrated security features
- Multiple audio interfaces
- High thermal efficiency
- Flexible power management

Axiomtek
www.axiomtek.com



BeagleBoard BeagleBone AI-64

The BeagleBone AI-64 features the Texas Instruments' AM572x Sitara, a dual-core ARM Cortex-A15 CPU and additional AI processing capabilities, making it ideal for edge AI applications. With a focus on machine learning, it provides a rich set of I/O options, including HDMI and dual Gigabit Ethernet ports. An open-source software environment encourages innovation and collaboration. The compact design and extensive GPIO support enable integration into various projects, from robotics to smart devices. Designed for reliability, the BeagleBone AI-64 is suitable for both educational and industrial applications.

- Dual-core ARM Cortex-A15 CPU
- Up to 1GB DDR3 RAM
- 16GB eMMC storage
- HDMI 2.0 output
- Dual Gigabit Ethernet ports
- 92 GPIO pins
- SPI, I²C, and UART interfaces
- Compact dimensions: 90mm x 60mm
- Low power consumption
- Extensive software support
- Designed for AI and machine learning applications
- Community-driven development resources
- Real-time clock support
- Integrated security features
- Multiple audio interfaces
- High thermal efficiency
- Ideal for robotics and smart devices
- Wide operating temperature range

BeagleBoard
www.beagleboard.org

DATASHEET URLS:

AAEON: https://www.aaeon.com/en/product/detail/developer-board-intel-11th-up-xtreme-i11

Axiomtek: https://www.axiomtek.com/Download/Spec/en-US/capa13r.pdf

BeagleBoard: https://docs.beagleboard.org/boards/beaglebone/ai/03-design-and-specifications.html

Datasheet: System-on-Module Technology

iWave Systems i.MX 8M Plus SoM

The i.MX 8M Plus SoM features a quad-core ARM Cortex-A53 CPU and integrated NPU for AI processing, making it suitable for multimedia and edge computing applications. With support for up to 8GB of LPDDR4 RAM, it provides robust performance for various tasks. The module includes HDMI 2.0, USB 3.0, and Gigabit Ethernet for versatile connectivity. Its compact 82mm x 50mm design ensures easy integration into space-constrained applications, while its low power consumption promotes energy efficiency. The i.MX 8M Plus is ideal for developers looking to implement AI and multimedia solutions in their projects.

- Quad-core ARM Cortex-A53 CPU
- Up to 8GB LPDDR4 RAM
- Up to 64GB eMMC storage
- HDMI 2.0 output
- USB 3.0 and PCIe support
- Gigabit Ethernet port
- Compact dimensions: 60mm x 25mm
- Low power consumption
- Wide operating temperature range
- Extensive software support
- Integrated NPU for AI processing
- Real-time clock support
- Multiple audio interfaces
- SPI, I^{2}C, and UART interfaces
- Ideal for multimedia applications
- Community-driven development resources
- High thermal efficiency
- Flexible power management

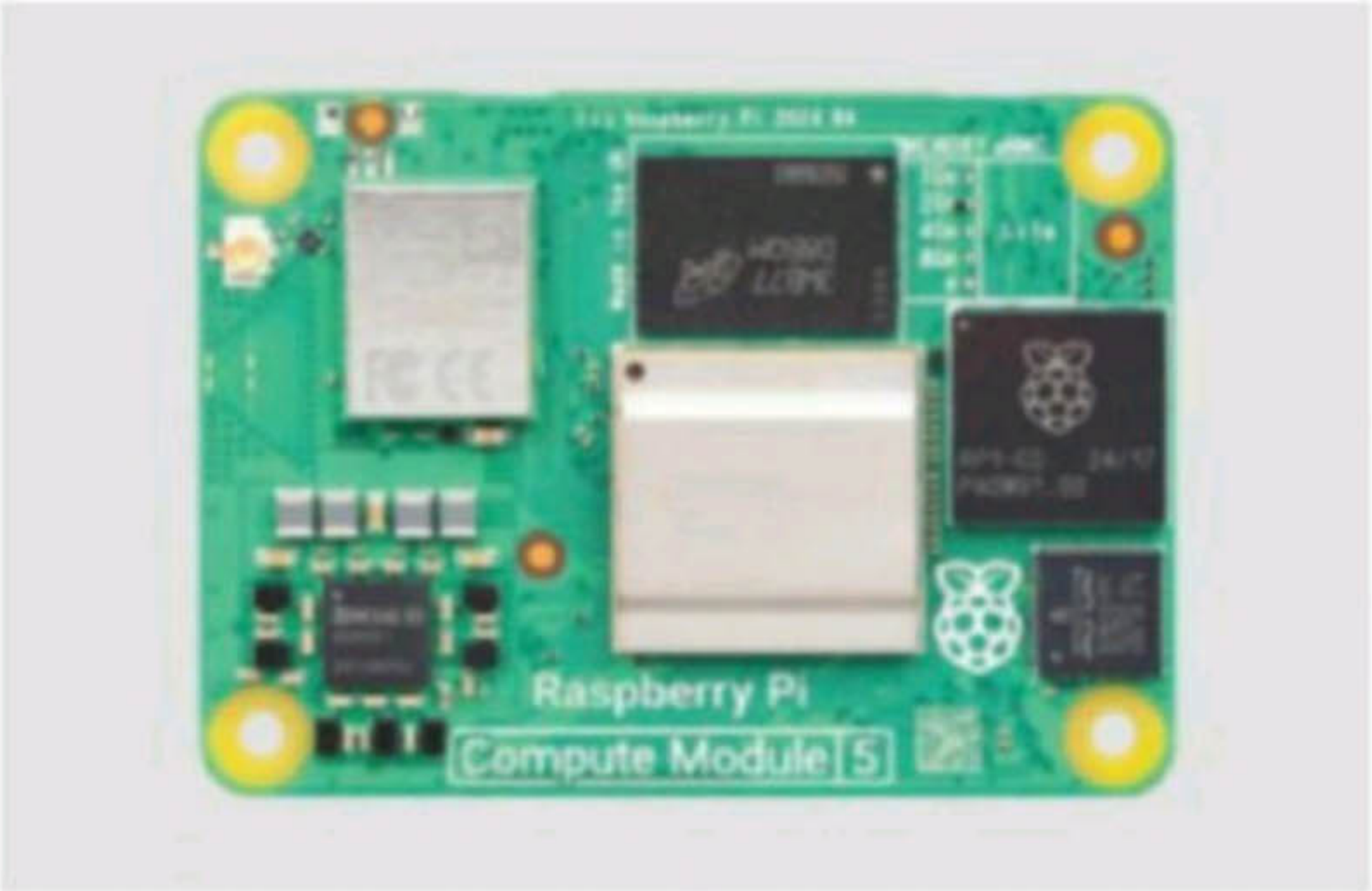
iWave Systems
www.iwavesystems.com

Nvidia Jetson Orin Nano

The Jetson Orin Nano is designed for AI applications, featuring a six-core ARM Cortex-A78 CPU and up to 8GB of LPDDR5 RAM. A 512-core NVIDIA Ampere architecture GPU with 16 tensor cores supporting up to 67 TOPS make it suitable for edge computing, robotics, and smart devices. The board supports multiple I/O options for high-performance multimedia applications. The Orin Nano designed for low power consumption while delivering high-performance for developers looking to AI in real-world applications.

- 6-core ARM Cortex-A78 CPU
- Up to 8GB LPDDR5 RAM
- 16GB eMMC storage
- HDMI 2.1 output
- USB 3.2 Type-C ports
- Gigabit Ethernet port
- M.2 Key M slot for NVMe SSD
- 40 GPIO pins
- SPI, I^{2}C, and UART interfaces
- Hardware video decoding support (H.265, H.264)
- Integrated AI accelerators
- Power management integrated circuit
- Real-time clock support
- Compact dimensions: 100mm x 80mm
- JetPack SDK
- Applications: Robotics, drones, AI applications
- Low power consumption design
- High thermal efficiency

Nvidia
www.nvidia.com



Raspberry Pi Compute Module 5

The Raspberry Pi Compute Module 5 features Broadcom's quad-core ARM Cortex-A76 CPU, and with RAM options up to 16GB. To enhance the capabilities of embedded systems, the CM5 provides dual HDMI outputs and improved connectivity options. The module retains compatibility with existing peripherals while expanding support for PCIe devices. Designed for a variety of applications, from industrial automation to DIY projects, the Compute Module 5 will offer a powerful platform for developers looking to create innovative solutions. Extensive community support ensures resources for rapid development and deployment.

- Quad-core ARM Cortex-A76 CPU
- Up to 16GB RAM
- Dual HDMI outputs
- USB 3.0 and PCIe support
- Gigabit Ethernet port
- GPIO pins for extensive connectivity
- Compact dimensions: 55mm x 40mm
- Low power consumption
- Wide operating temperature range
- Extensive software support
- Community-driven development resources
- High-performance multimedia capabilities
- Integrated security features
- Real-time clock support
- Multiple audio interfaces
- Flexible power management
- Support for various display interfaces

Raspberry Pi
www.raspberrypi.com

DATASHEET URLS:

Iwave Systems: https://d3pwr7ye9vqskz.cloudfront.net/wp-content/uploads/2023/08/iW-RainboW-G40M_i.MX_8M_Plus_SMARC_SOM-HardwareUserGuide-R2.0-REL1.0.pdf

Nvidia: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/#tech-specs

Raspberry Pi: https://datasheets.raspberrypi.com/cm5/cm5-product-brief.pdf

SolidRun HoneyComb LX2K

SolidRun's HoneyComb LX2K features a 16-core ARM Cortex-A72 CPU, designed for high-performance applications in networking and edge computing. With support for up to 32GB of DDR4 RAM, it provides substantial processing power for data-intensive tasks. The module includes multiple I/O options, including PCIe and Gigabit Ethernet. Its compact design and energy efficiency make it suitable for various applications, from telecommunications to industrial automation. The HoneyComb LX2K is optimized for scalability, enabling creation of robust solutions tailored to specific needs.

- 16-core ARM Cortex-A72 CPU
- Up to 32GB DDR4 RAM
- Up to 64GB eMMC storage
- Dual Gigabit Ethernet ports
- PCIe expansion slots
- Compact dimensions: 100mm x 100mm
- Low power consumption
- Wide operating temperature range
- Extensive software support
- Designed for networking and edge computing
- Real-time clock support
- Integrated security features
- High thermal efficiency
- Multiple audio interfaces
- Ideal for industrial automation
- Flexible power management
- High-performance data processing capabilities

SolidRun
www.solid-run.com

Toradex Colibri iMX8M Plus

The STM32L0 series is STMicroelectronics' entry-level range of 32-bit MCUs designed for ultra-low power consumption levels as low as 49µA/MHz with external DC/DC. With dynamic voltage scaling, the STM32L051x6 with an ARM-based Cortex-M0+, with 64KB Flash, 8KB SRAM, 2KB EEPROM, and an ADC is aimed at energy harvesting, smoke detectors, gas and water meters and consumer devices with advanced battery management and fast charging.

- Quad-core ARM Cortex-A53 CPU
- Up to 8GB LPDDR4 RAM
- Up to 64GB eMMC storage
- USB 3.0 and USB 2.0 ports
- HDMI 2.0 output
- Gigabit Ethernet port
- 40 GPIO pins
- SPI, I²C, and UART interfaces
- Compact dimensions: 67.6mm x 36.5mm
- Wide temperature range: -40°C to +85°C
- Integrated GPU for multimedia processing
- Low power consumption
- Real-time clock support
- Security features for data protection
- Extensive software support
- Ideal for industrial and IoT applications
- Community-driven resources
- Multiple audio interfaces

Toradex
https://www.toradex.com

Variscite DART-MX8M Plus

The Variscite DART-MX8M Plus pushes a quad-core ARM Cortex-A53 CPU with robust performance for multimedia applications. Up to 8GB of LPDDR4 RAM and 64GB eMMC storage powers high-performance for industrial and consumer embedded systems. Interfaces for HDMI 2.0, USB 3.0, and Gigabit Ethernet offer versatile connectivity and a compact footprint is suitable for space-constrained applications, while low power consumption ensures efficient operation. demands are many and supply is finite.

- Quad-core ARM Cortex-A53 CPU
- Up to 8GB LPDDR4 RAM
- Up to 64GB eMMC storage
- HDMI 2.0 output
- USB 3.0 and USB 2.0 ports
- Gigabit Ethernet port
- MIPI DSI and CSI interfaces
- Compact dimensions: 50mm x 25mm
- Low power consumption
- Wide temperature range: -40°C to +85°C
- Integrated GPU for multimedia processing
- Multiple audio interfaces
- Real-time clock support
- Security features for data protection
- Ideal for multimedia and IoT
- SDIO/MMC
- 1x PCIe v3.0
- Resistive touch controller
- Serial interfaces (ECSPI, FlexSPI, I²C, UART, CAN, JTAG)
- Single power supply: 3.3V
- Industrial temperature range: -40°C to 85°C

Variscite
https://www.variscite.com

DATASHEET URLS:

SolidRun: https://www.solid-run.com/wp-content/uploads/2023/01/HoneyComb-LX2-Datasheet-2023.pdf

Tordex: https://docs.toradex.com/108784-verdin_imx8m_plus_datasheet.pdf

Varascite: https://www.variscite.com/wp-content/uploads/2020/10/VAR-SOM-MX8M-PLUS_Datasheet.pdf

Modern Embedded Insights

CMake the Most of Software Development

Modern Build Systems Support Scalability

By
Jacob Beningo

Discover how modernizing your build system with CMake can revolutionize embedded software development, offering unparalleled flexibility and efficiency. Dive into this article to explore the transformative benefits of CMake's cross-platform capabilities and how it integrates seamlessly with contemporary development practices.

Many embedded software developers today cannot take advantage of modern development techniques like DevOps, Test-Driven Development, and simulation because they write software using their microcontroller vendor's IDE. While these integrated environments are convenient for getting a project started quickly, they don't lend themselves well to building multiple configurations; they often build slowly and can create productivity headaches.

Embedded software development doesn't need to be that way. There's a better way that will allow you to leverage modern development techniques and practices while still having the benefits of an integrated IDE: create your own modern CMake build system. I know this sounds time-consuming and difficult, but I will show you how easy it can be.

Those who want to jump straight to the solution can download a completed version at a link in the article resources. [1]

WHAT'S A BUILD SYSTEM?

A build system is a set of tools and processes used to automate converting source code into executable programs. It includes:

- Cross-compilation
- Configuration management
- Automation
- Integration with development tools
- Resource handling

A typical build system will look like **Figure 1**.

The application comprises drivers, an operating system, some middleware, and the application itself. These are cross-compiled using Make [2] with the target compiler, possibly GCC, and then linked into a final binary that runs on the target microcontroller. There is typically just a single build configuration, which forces you to run the code directly on the target and only on the target.

The problem with the traditional build system is that it doesn't scale well or provide a developer with the right tools to build a modern system. You need a build system that supports a modern workflow to succeed today. But what does that build system look like?

DESIGNING A BUILD SYSTEM

A modern build system doesn't just support a single target; it supports a flexible and scalable architecture with multiple build targets. There are a minimum of five targets that the modern build system should support:

- Debug
- Release
- Test
- Simulation
- Analysis

The debug and release builds are similar to the traditional, single-target builds. They build the software to run on your microcontroller in either an unoptimized (debug) or optimized (release) manner. Different compilation flags are applied to simplify debugging your software or optimize it for a production build.

The test build is all about enabling test automation and DevOps. Using an automated unit test harness is a critical technique to ensure you are writing robust, tested code. The test build will run unit tests, regression tests, and maybe even hardware-in-the-loop tests. The idea is that every time you commit your code, the test build will automatically run and make sure that the changes you made to the code didn't break anything.

Today's embedded systems have reached a level of complexity and sensor integration that lends itself well to using simulation. Simulation is a technique that allows developers to run application code off-target to prove that it works correctly without having to run it on hardware. It's a powerful technique and one that every developer should be supporting.

Finally, the analysis build scans your code and performs metric measurements so that you can have a scoreboard to tell you how good or bad your software is. Cyclomatic Complexity is an example of a metric often checked in these builds. The idea is that the more complex your functions are, the greater the risk of a bug in that code. Minimizing complexity allows you to keep code simple, reducing the chances of bugs.

With this knowledge, you might design a modern build system that looks like **Figure 2**. As you can see, it supports all these builds along with a software architecture and build system tools that allow for flexibility and reuse of the developed code. Many developers today write code to be used in more than one product, and a build system like this enables them to reuse their code across multiple products. I often call this the development of a platform.

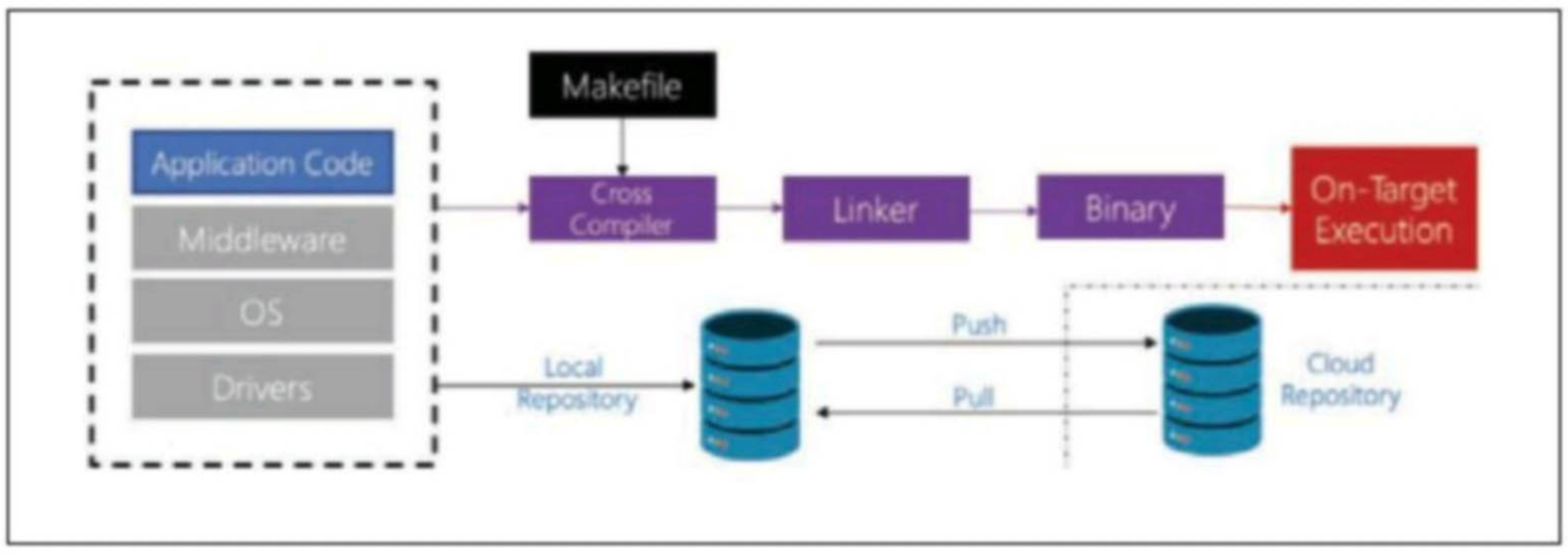

Application Code
Middleware
OS
Drivers
Makefile
Cross Compiler
Linker
Binary
On-Target Execution
Local Repository
Push
Pull
Cloud Repository

FIGURE 1
The traditional build system has a single target that builds for the microcontroller.

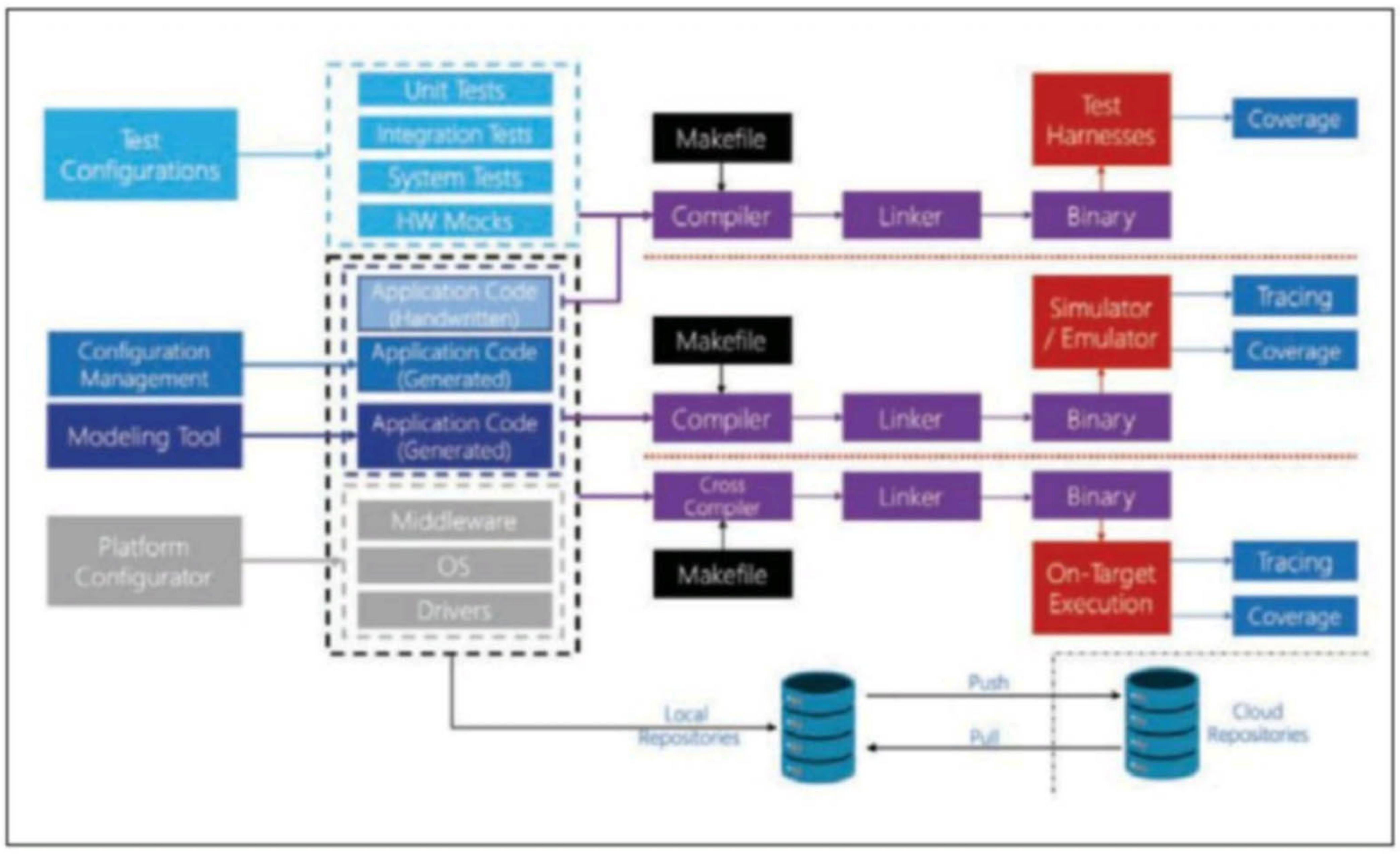

Test Configurations
Unit Tests
Integration Tests
System Tests
HW Mocks
Configuration Management
Modeling Tool
Platform Configurator
Application Code (Handwritten)
Application Code (Generated)
Application Code (Generated)
Middleware
OS
Drivers
Makefile
Compiler
Linker
Binary
Test Harnesses
Coverage
Makefile
Compiler
Linker
Binary
Simulator / Emulator
Tracing
Coverage
Cross Compiler
Linker
Binary
Makefile
On-Target Execution
Tracing
Coverage
Local Repositories
Push
Pull
Cloud Repositories

FIGURE 2
A modern build system is flexible, scalable, and can support multiple build targets, including test, simulation, and code analysis.

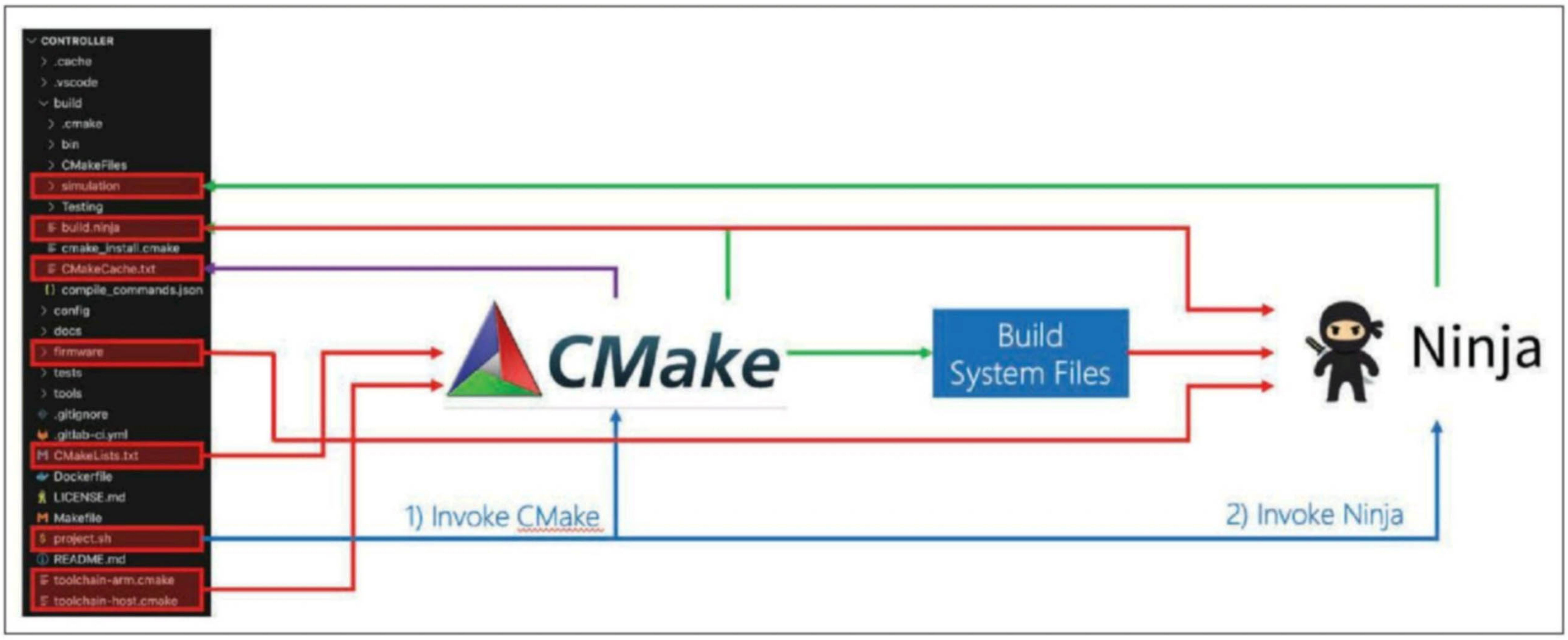

CONTROLLER
.cache
.vscode
build
.cmake
bin
CMakeFiles
simulation
Testing
build.ninja
cmake_install.cmake
CMakeCache.txt
compile_commands.json
config
docs
firmware
tests
tools
.gitignore
.gitlab-ci.yml
CMakeLists.txt
Dockerfile
LICENSE.md
Makefile
project.sh
README.md
toolchain-arm.cmake
toolchain-host.cmake
CMake
Build System Files
Ninja
1) Invoke CMake
2) Invoke Ninja

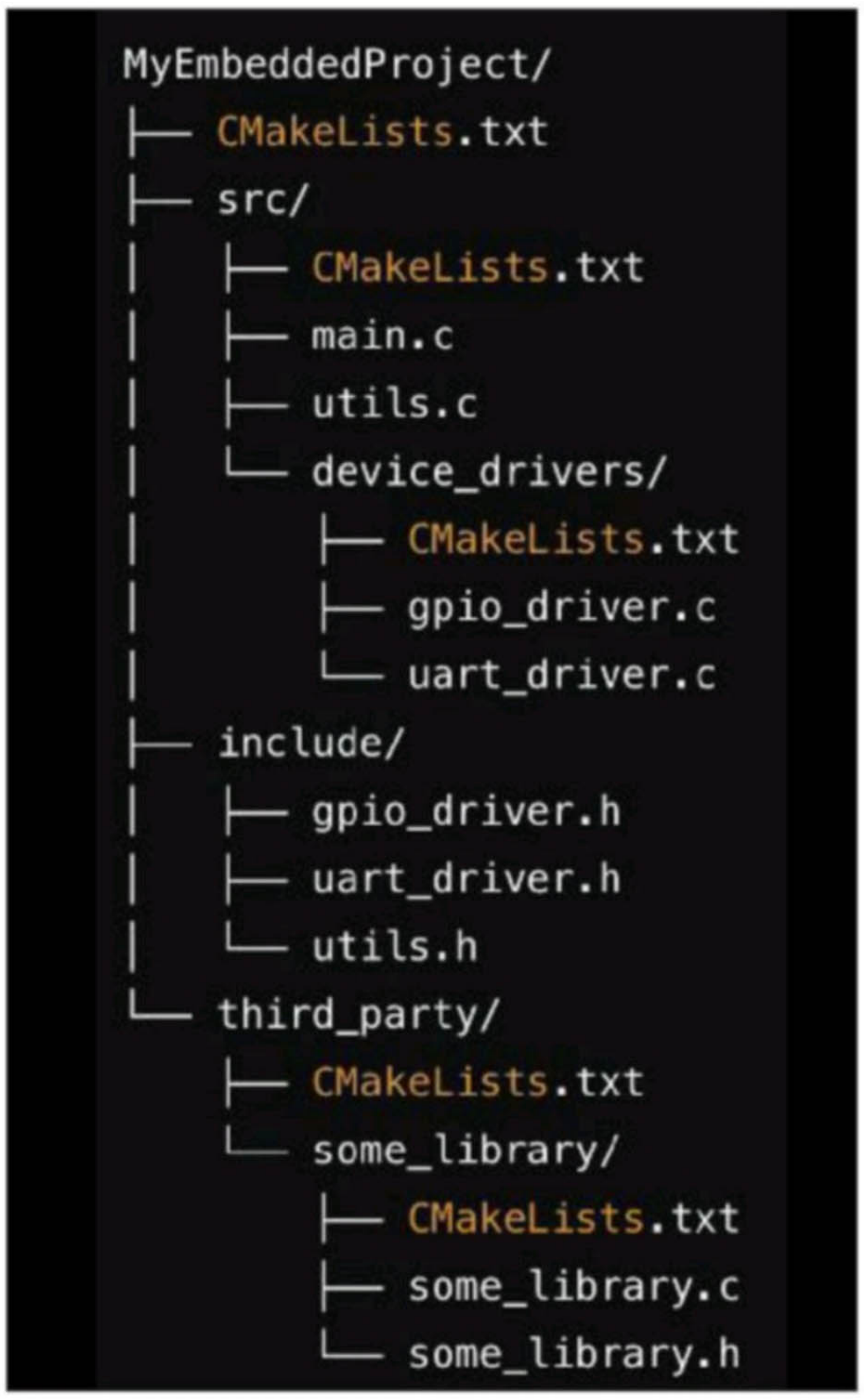
FIGURE 3
CMake uses CMakelists.txt files and toolchain files to generate the build system files that are used by other tools to compile the software into the target binary.

Now that we know what a modern build system should look like, let's discuss how to implement the design using CMake.

BUILD A SYSTEM GENERATOR

CMake is a cross-platform, open-source build system generator. It simplifies the process of managing build processes in a compiler-independent manner, making it ideal for projects that need to operate across different systems. It brings:

- Cross-platform support
- Configurability
- Modularity
- Extensive language support
- Toolchain integration



```
MyEmbeddedProject/
├── CMakeLists.txt
├── src/
│   ├── CMakeLists.txt
│   ├── main.c
│   ├── utils.c
│   └── device_drivers/
│       ├── CMakeLists.txt
│       ├── gpio_driver.c
│       └── uart_driver.c
├── include/
│   ├── gpio_driver.h
│   ├── uart_driver.h
│   └── utils.h
└── third_party/
    ├── CMakeLists.txt
    └── some_library/
        ├── CMakeLists.txt
        ├── some_library.c
        └── some_library.h
```

FIGURE 4
In complex or large projects, CMakelists.txt files are included in nearly all directories to inform CMake of all the compiler details and settings for that specific set of software.

```
cmake_minimum_required(VERSION 3.10)
project(controller C CXX ASM)
# Set target
set(TARGET controller)
# Set the output directory for the executables
set(EXE_DIR ${CMAKE_BINARY_DIR}/bin)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${EXE_DIR})
# Initialize lists for sources and include directories
set(ALL_C_SOURCES “”)
set(ALL_CPP_SOURCES “”)
set(ALL_ASM_SOURCES “”)
set(ALL_INCLUDE_DIRS “”)
# Add subdirectories
add_subdirectory(firmware/app)
add_subdirectory(firmware/bsp)
add_subdirectory(firmware/drv)
add_subdirectory(firmware/hal)
add_subdirectory(firmware/lib)
```

LISTING 1
A CMakelists.txt file contains commands, variables, and conditional logic required to prepare the build system files for compiling the application.

Instead of writing a bunch of Makefile scripts, which can get complicated quite quickly, using CMake provides a modular programming language to configure how you want your build system to be generated. Behind the scenes, CMake can take what you provide and generate Make or Ninja files that are then executed to build the final targets. **Figure 3** shows an example how you might use CMake with Ninja to compile an embedded application. You can see that CMake's job is to generate the build system files that are used by Ninja to compile the final binaries.

How you use CMake will depend on the size and complexity of the project that you are building. If you are working on a simple project that is relatively self-contained, then you would create a single top-level CMakelists.txt file to manage the build system generation. CMakelists.txt contains all the commands, variables, and conditional logic to create the build system files.

In a more complex, larger project where a lot of reuse will occur, a single top-level CMake file won't provide enough flexibility. Instead, a CMakelists.txt file is provided in all the projects subdirectories. A high-level CMakelists.txt file is then used to set global settings, directories to search, etc. When each directory is traversed, the local CMakelists.txt file is read for the detailed settings on how to compile the source in that directory. **Figure 4** shows an example how a project might look in this configuration.

UNDERSTANDING "CMAKELISTS.TXT FILES"

At first exposure, the CMakelistsl.txt file can seem a little overwhelming, but once you look closely you'll see that the syntax is relatively straight forward. For example, in **Listing 1**, you can see the start of a CMakelists. Every CMakelists.txt file starts with `cmake_minimum_required` to tell the build system what the minimum version of CMake required to generate the build system. CMake, like any other piece of software, evolves with time and is constantly adding new features and easier ways to do things.

The line project (controller C CXX ASM) is also important. It's a command that gives our project the name 'controller' and tells CMake what programming languages we'll be using.

```
# Add the executable target
add_executable(${PROJECT_NAME} ${ALL_C_SOURCES} ${ALL_CPP_SOURCES} ${ALL_ASM_SOURCES})
set_target_properties(${PROJECT_NAME} PROPERTIES OUTPUT_NAME ${PROJECT_NAME}.elf)
# Set the include directories for the target
target_include_directories(${PROJECT_NAME} PRIVATE ${ALL_INCLUDE_DIRS})
```

LISTING 2
Configuring the project output using CMake.

In this example, you can see that we are naming the project controller and that we'll be creating a mixed language project that uses C, C++, and assembly language.

You can also see in Listing 1 that we use the command set to create variables that hold a list of the files we want to compile. We will add to those files later, along with the desgination of where we want to generate our final output file.

CREATING A BUILD SYSTEM

Setting the programming language and setting some directories is not enough to create a build system. We need to set the target, the target properties and where the final target will be placed when the compilation process is complete. **Listing 2** shows how we do all those things in CMake. Once again, we use a few CMake commands and variables to tell CMake what we need it to do.

One final thing we need to get a basic, modern build system up and running using CMake is to set our linker options. **Listing 3** shows a few additional statements that we add to CMakelists.txt that pull in our linker options.

If you've been looking carefully, you might realize that this CMakelists.txt file seems to use variables to reference values that aren't defined in CMakelists.txt! How does CMake know what settings to use?

To answer that question, we need to talk about toolchain files.

CUSTOMIZING TARGETS AND TOOLCHAINS

A toolchain file in CMake is a crucial configuration file that specifies the compiler and tools to be used for compiling and cross-compiling various projects. The toolchain file clearly specifies the target system, such as Linux, Darwin, Windows, Arm, and so forth. It also specifies which compiler is used and the exact path to the compiler. For example, if you are going to build your application for a specific simulation build, you might have a toolchain file similar to Listing 4. You can see that in the simulation toolchain file, we want to build specifically for Darwin (MacOS) and use the GCC compiler.

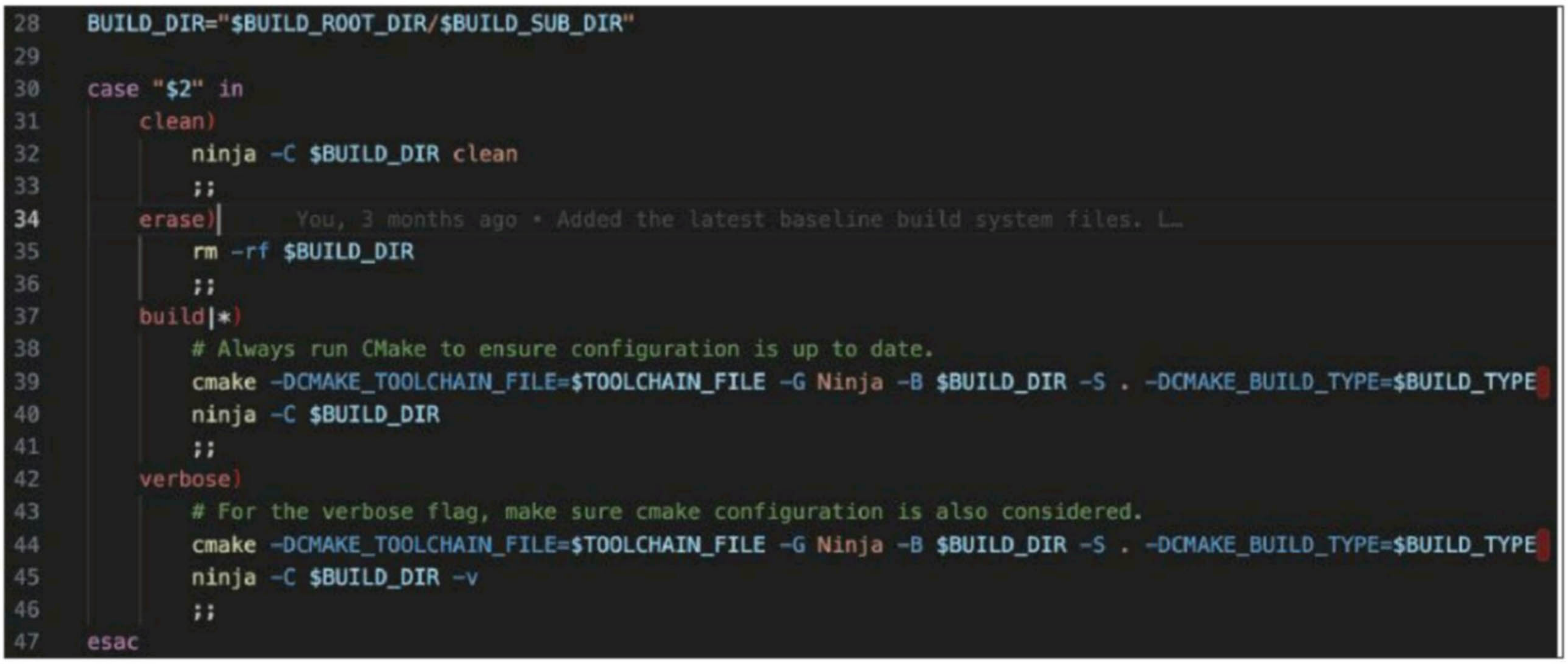
The toolchain file can contain any build-specific information that might change from one target to the next. For example, if I'm building for an Arm Cortex-M microcontroller, I would specifically use the arm-none-eabi-gcc compiler. However, as you saw, for my

```
# Linker options from the toolchain file
target_link_options(controller PRIVATE
${TOOLCHAIN_LINKER_OPTIONS})
# Libraries from the toolchain file
target_link_libraries(${TARGET} ${TOOLCHAIN_LIBRARIES})
```

LISTING 3
CMake commands to set the toolchain linker and library options.

```
# Use the system’s default C and C++ compilers
set(CMAKE_SYSTEM_NAME Darwin) # Adjust this as necessary (e.g., Windows, Darwin for macOS, Linux)
# Optionally specify the compiler (GCC/Clang) if the default selection needs to be overridden
set(CMAKE_C_COMPILER gcc)
set(CMAKE_CXX_COMPILER g++)
# Add any simulation-specific preprocessor definitions
add_compile_definitions(SIMULATION)
# Specify any host-specific compiler flags if necessary
set(COMMON_FLAGS “-O2 -g”)
set(CMAKE_C_FLAGS “${CMAKE_C_FLAGS} ${COMMON_FLAGS}” CACHE STRING “” FORCE)
set(CMAKE_CXX_FLAGS “${CMAKE_CXX_FLAGS} ${COMMON_FLAGS}” CACHE STRING “” FORCE)
# Set the C standard and C++ standard that your project requires
set(CMAKE_C_STANDARD 17)
set(CMAKE_CXX_STANDARD 17)
```

LISTING 4
An example of a tool chain file for developers building and application for a simulation build.



```
BUILD_DIR="$BUILD_ROOT_DIR/$BUILD_SUB_DIR"

case "$2" in
    clean)
        ninja -C $BUILD_DIR clean
        ;;
    erase)      You, 3 months ago • Added the latest baseline build system files. L…
        rm -rf $BUILD_DIR
        ;;
    build|*)
        # Always run CMake to ensure configuration is up to date.
        cmake -DCMAKE_TOOLCHAIN_FILE=$TOOLCHAIN_FILE -G Ninja -B $BUILD_DIR -S . -DCMAKE_BUILD_TYPE=$BUILD_TYPE
        ninja -C $BUILD_DIR
        ;;
    verbose)
        # For the verbose flag, make sure cmake configuration is also considered.
        cmake -DCMAKE_TOOLCHAIN_FILE=$TOOLCHAIN_FILE -G Ninja -B $BUILD_DIR -S . -DCMAKE_BUILD_TYPE=$BUILD_TYPE
        ninja -C $BUILD_DIR -v
        ;;
esac
```

FIGURE 5
Using a script like project.sh or devManager.sh can help simplify selecting build targets and compiling your project.

simulation build, I would simply use the local machine's gcc compiler.

Toolchain files provide you with the flexibility to change build settings from one target build to the next. They are extremely powerful. You can see from Listing 4 that we can even have different compiler flags, defines, linker settings, and so forth. It's completely customizable allowing us to support multiple targets and builds without having to change our source code. You simply build with a different toolchain file and those settings are substituted into our CMakelists.txt file during the build.

MODERN SYSTEM BUILDING

Once you design and set up your modern build system, you'll still need to build your project. There are several ways that you could compile a project. First, you can directly invoke CMake at the command line using a command like:

```
cmake -G Ninja -B /build -S .
   -DCMAKE_BUILD_TYPE=simulation
   -DCMAKE_TOOLCHAIN_
   FILE=toolchain-host.cmake
```

This command is telling CMake that we want to generate the necessary build system files for Ninja that are specifically for a simulation build using the toolchain-host.cmake file. The output from CMake will be conveniently placed in the /build directory.

Invoking CMake in this way will generate our build system files but it won't compile our project. To do that, we also need to invoke Ninja using the command:

```
ninja -C /build
```

Once we do that, Ninja will read in the files output by CMake, invoke our compiler and generate the output binary for our build.

SCRIPTS SIMPLIFY

Using the command to build our various targets from the command line is a bit exhausting. I don't know about you, but I don't want to memorize that command string. I know that I can just use the command recall in my terminal to execute the command, but if I give my build system to a colleague, they won't have access to it.

One solution to enable easy building and target selection is to use a build script. I like to create a build script name project.sh or devManager.sh to manage the various build targets. The build script can be broken up into several parts that you to:

1) Select the target.
2) Select whether you want to build or run that target.

There are quite a few details that can go into creating the script, but let's just look at what the details of how we can simplify the build. You can see in **Figure 5** a snippet from the project.sh script that manages the build string. You can see that instead of specifying the toolchain file, build directories, and build types, we instead replace them with a variable.

The variable is set dynamically based on the specific command that needs to be executed. For example, if I want to build

Additional materials from the author are available at:
www.circuitcellar.com/article-materials
Resources [1] to [2] as marked in the article can be found there.

specifically for simulation, at the terminal I might type:

```
./project.sh simulation build
```

That command will be taken by my script and the correct build strings generated. In this case, the following substitutions would occur:

- `$TOOLCHAIN_FILE => toolchain-host.cmake`
- `$BUILD_DIR => /build/simulation/`
- `$BUILD_TYPE => simulation`

Using a script in this manner can make your build system far more flexible and help free up mental bandwidth for more important things like writing code.

CONCLUSIONS

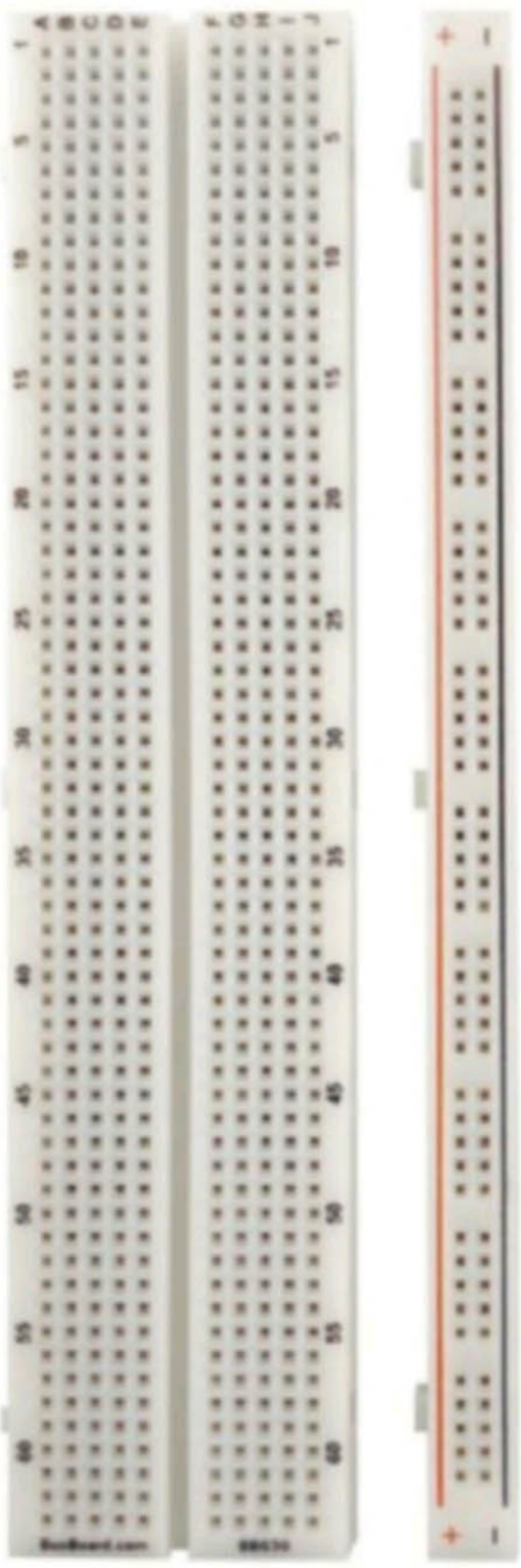
There are a lot of advantages to using CMake as the foundation for your modern embedded build system. As you've seen, it's flexible and can easily scale to include all the targets that a modern build system should support. More importantly, this flexibility allows you to create the build targets that make modern development techniques like DevOps, Test-Driven Development, and simulation available to your development cycle.

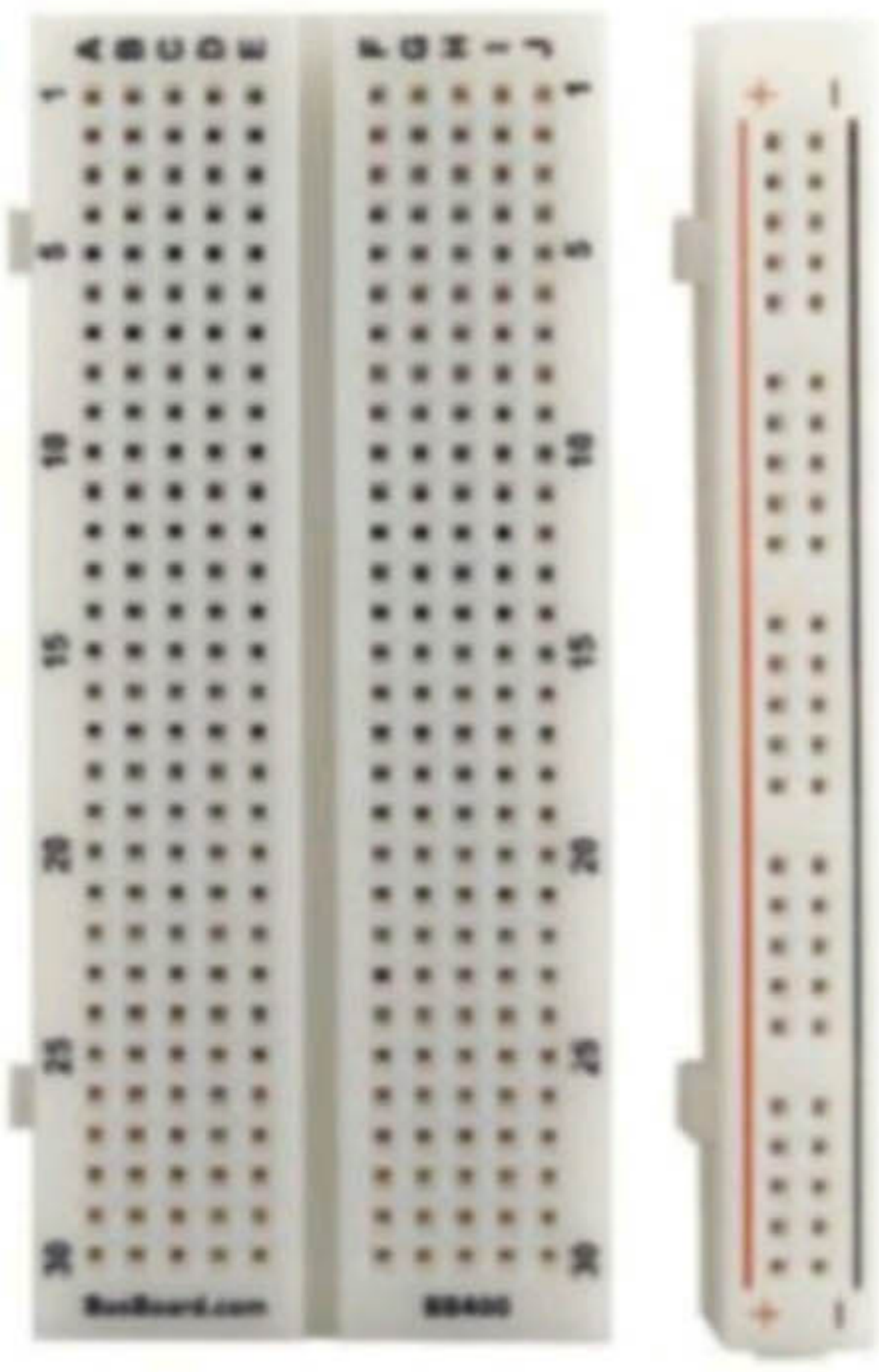
It's time to break the shackles of your existing build system and design or adopt your own. If you think that it's too complicated, don't worry. You can download the example for this article and use that as your baseline. Once you've reviewed and understand how it works, you can adapt it for your own project needs.

When you take this build system and use it with Visual Studio Code, you not only have a modern build system but also an integrated development environment that you can customize and scale for years into the future. I hope that you can now clearly see that this is a better, more flexible way to develop embedded software.

ABOUT THE AUTHOR

Jacob Beningo is an embedded software consultant and founder of the Embedded Software Academy, where he helps engineers master modern practices to design efficient, scalable embedded systems. He helps developers stay ahead in an evolving industry with hands-on training, expert insights, and strategies. He's consulted and trained companies such as Arm, General Motors, NASA, and others, completing more than 200 projects. Jacob has published books on embedded software development techniques. He holds three degrees, including a master's in space systems engineering from the University of Michigan. You can learn more about Jacob's work at www.beningo.com.





Modular BreadBoards

Mix and match our BreadBoard parts to perfectly fit your project

Start your prototyping with one of our many full-height (*BB830M, BB1460, BB1560M*) or half-height (*BB400M*) breadboards and easily detach and rearrange the power rails and circuit areas to build the perfect layout.

Easily change your layout and add additional circuit area (*BB630, BB300*) and power rail (*BB100R, BB50R*) components as your project grows.

Purchase at your trusted retailer

MOUSER ELECTRONICS DigiKey amazon.com

BPS™ BusBoard Prototype Systems

www.BusBoard.com/BB

Start to Finish

Switching Regulator Efficiency

Minimize Power Consumption in Battery-Operated Systems

By
Stuart Ball

A switching voltage regulator converts input DC voltage into a desired output DC voltage. This month, Stuart delves into the factors that influence the efficiency of switching regulators, and why they are crucial for embedded applications. From understanding the role of inductors and transistors to exploring practical examples, this article offers valuable knowledge for anyone interested in optimizing power supply designs.

In a lot of embedded applications, you will use a single power supply, either from the AC line or from batteries. But in some cases, you will have an application that needs more than one voltage. For example, you might have a battery-operated system that needs a 9V or 12V supply voltage for an op amp that is sensing something from a sensor. Or you might need to illuminate an LCD backlight using a voltage that is higher than the supply voltage. You might have a device that operates from a 6V battery (such as four AAA cells), but that has to power a 3.3V MCU.

In all these cases, you need a voltage regulator—a device that produces a constant output voltage to power an electronic circuit. In each instance, you can use a switching power supply to generate the needed voltage. A switching regulator, whether step-down or step-up, uses switching techniques to produce a regulated output voltage. But, especially in battery-operated equipment, you want the power supply to be as efficient as possible, so as to minimize the power it uses.

In my column on "Voltage Regulators" (*Circuit Cellar* #387, October 2022) [1], I looked at voltage regulators, power, and efficiency, and I touched on switching regulators. This month, I want to narrow specifically to the things that affect the efficiency of switching regulators.

SWITCHING SUPPLY TYPES

Switching supplies come in three basic configurations:

- AC-to-DC converters change AC input to a DC output. An example is the power supply in a desktop computer. I won't be covering those here.
- Step-up or boost converters change a DC input voltage to a higher output voltage. An example would be the aforementioned switcher that converts the 3.3V MCU supply voltage to 6V to power an LCD backlight.
- Step-down or buck converters that convert a DC voltage to a lower DC voltage. An example would be a power supply that takes a 9V input from a "wall wart" power supply and produces 3.3V for an MCU.

Switching converters can also produce a negative output from a positive input or the other way around. This can be a step-up or step-down converter; the defining characteristic is that the output voltage is the opposite polarity of the input voltage. A switcher to power an RS-232 driver or the negative supply for an opamp would be examples.

Figure 1 shows the configurations of the three types of DC-DC switchers. These simplified diagrams illustrate the basic mechanism for generating a switched DC voltage. (The control and voltage regulation parts are not shown.)

All three of these use the same basic principle. An electronic switch is closed, providing current to an inductor. When the switch is opened, the energy stored in the inductor is dissipated through a rectifier to produce an output voltage.

Any inductive device, whether a coil in a switcher, the coil in a relay, a solenoid, or a motor winding, will store energy when current is passed through it. If the current is suddenly removed, energy conservation says the energy in the inductor has to go somewhere. In some applications, the energy is dissipated in a snubber circuit to protect the electronics. In a switching power supply, however, the voltage is used to drive the output voltage. The voltage generated when the inductor energy is dissipated is often called "flyback" or "kickback" voltage. The term flyback dates back to the days when televisions had CRTs instead of LEDs.

The waveforms for the inductor voltage for the three types of switchers also are shown in Figure 1. In a real switcher, the switch is a transistor that is controlled by circuitry external to the diagram.

In the top diagram, which shows a boost converter, the switch grounds one end of L1 when closed. The current builds in L1, storing energy, until the switch is opened. At that point, the driven end of L1 rises, generating the flyback voltage. The voltage is rectified through D1 and filtered with C1 to produce a DC output voltage that is higher than the original supply voltage.

The second diagram is a step-down converter, in which the switch connects the driven end of L2 to V+. When the switch opens, the energy charges C2, producing a voltage lower than V+.

The third diagram is a negative voltage generator. It functions similarly to the boost converter, except that one end of the inductor is tied to ground instead of V+. The driven end is switched to V+. When the switch opens, voltage V3 goes negative. This is exactly what happens in the boost converter, except that the polarity is reversed. The negative voltage is rectified by D3 and filtered by C3. The result is a negative DC voltage at Vout. Note that if C3 is an electrolytic capacitor, the positive terminal is grounded.

You can build positive and negative boost converters using capacitors as the storage elements instead of inductors, but that's outside the scope of this article.

WHAT IS EFFICIENCY?

The efficiency of any power supply is the output power divided by the input power.

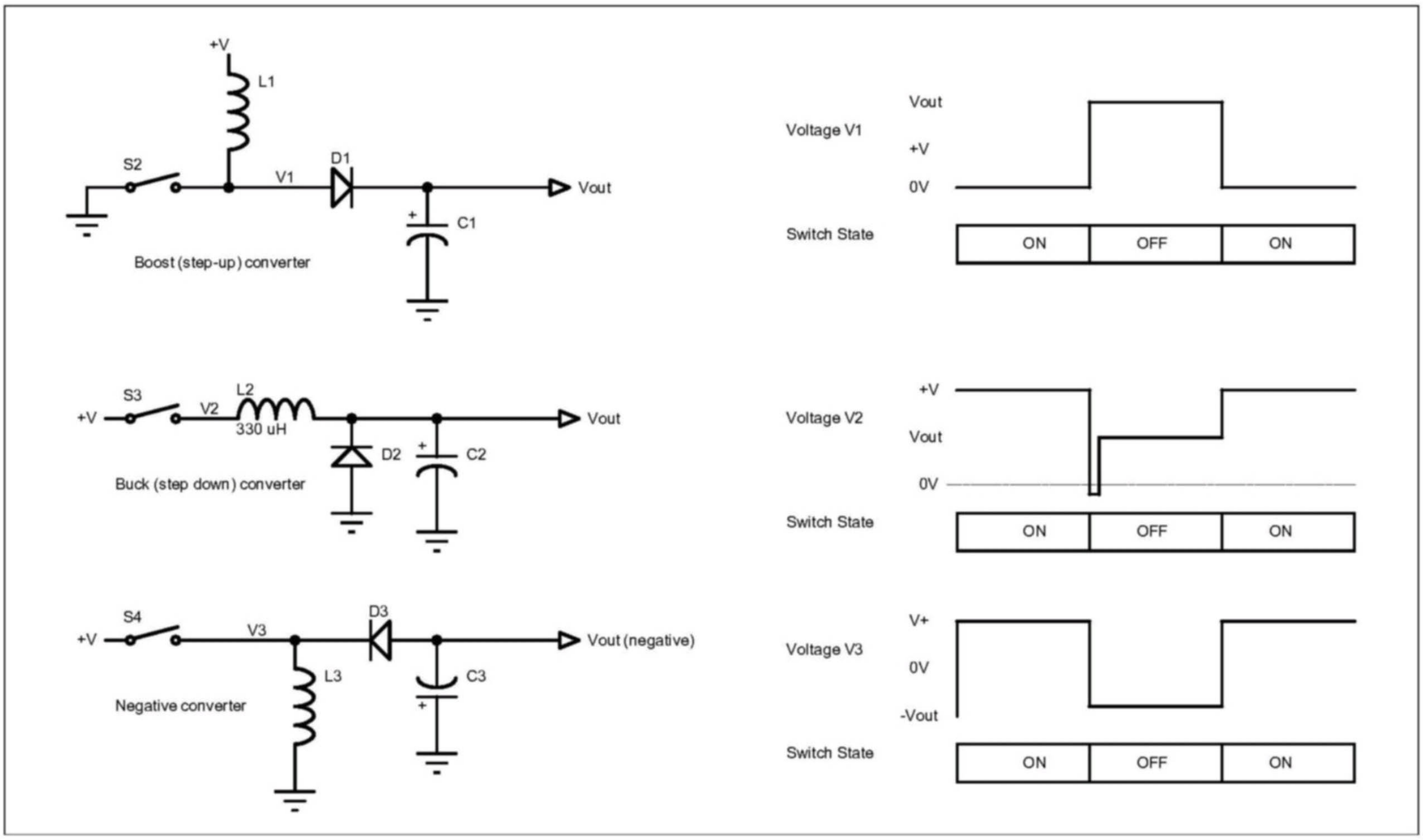


FIGURE 1
DC-DC switching configurations. Top: step-up (boost) converter. Middle: step-down (buck) converter. Bottom: negative voltage generator.

The output power will always be less than the input power, so efficiency will always be less than 100%. In the simple case, where the output load is a pure resistance, the output current is the output voltage (Vo) divided by the resistance, and the output power is Vo times the output current.

The input power is the input voltage multiplied by the input current. In a dynamic application, where the load varies, both input and output power will vary with the load current.

Any power supply efficiency less than 100% means that electrical current is being dissipated as heat — something is getting warmer than it otherwise would be. Since no power supply is 100% efficient, the goal is to get as close to 100% as practical, within the design parameters of the circuit (size, cost, power, and so on).

INDUCTOR CHARACTERISTICS

Figure 2 shows a simple switching converter. This takes a 5V input and produces a boosted voltage such as 9V or 12V that might be used to power an op amp or a backlight LED in an embedded design. It doesn't produce much current, and is just for demonstrating some principles. By using a fixed resistive load, I can directly measure the maximum power the circuit can produce by measuring the voltage across the load. In a real application, you would probably use a dedicated switching regulator IC. However, you might use a simple circuit like this one, if you've already got an MCU in the design and the appropriate outputs are available. You would have to regulate the output voltage, probably using a Zener for something this simple.

IC U1 produces a PWM output on the OCR1A output. This drives transistor Q1, which in turn drives the coil. Switch S1 is a quadrature rotary encoder with a pushbutton feature on the shaft. The encoder adjusts the PWM frequency from about 5kHz to about 50kHz. The encoder also adjusts the duty cycle from 10% to 90%. The firmware toggles between frequency and duty cycle mode by pressing the shaft of the switch. The OCR1A output drives a 2N2222 transistor or a 2N7000 MOSFET transistor through resistor R2. I socketed the transistor so I can switch between the 2N2222 and the 2N7000.

This isn't intended to be any kind of project; rather, it's just a tool to demonstrate some basic principles. But if you want to reproduce it, the code for the MCU is available on the *Circuit Cellar* Article Code and Files webpage. It's fairly simple, so it's all in one C file.

SWITCHER EFFICIENCY FACTORS

The primary characteristics of the inductor that affect switcher efficiency are the saturation current and the resistance.

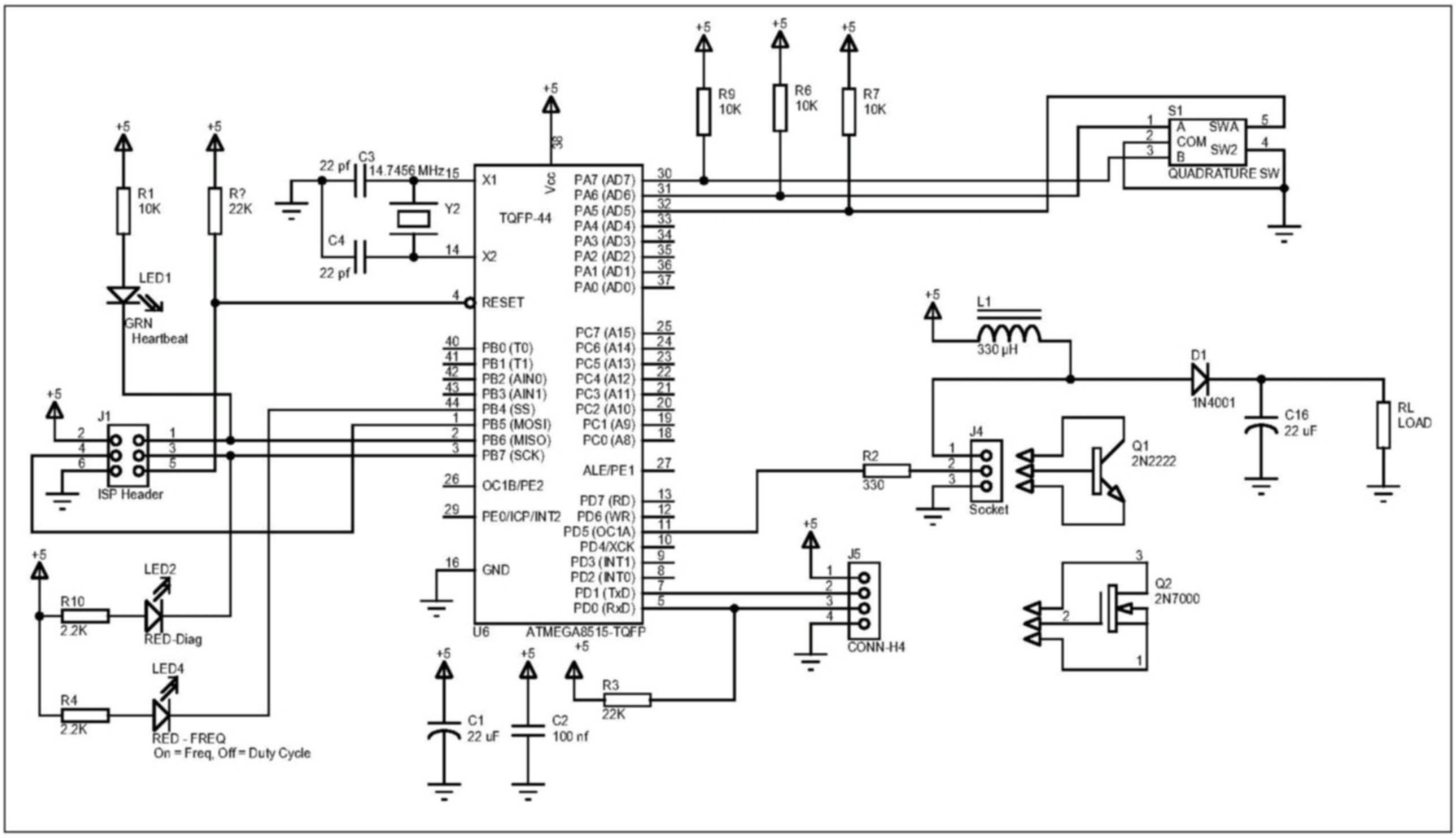

+5
R1
10K
R?
22K
LED1
GRN
Heartbeat
J1
ISP Header
R10
2.2K
LED2
RED-Diag
R4
2.2K
LED4
RED - FREQ
On = Freq, Off = Duty Cycle
C3
22 pf
14.7456 MHz
Y2
C4
22 pf
X1
X2
RESET
Vcc
TQFP-44
PA7 (AD7)
PA6 (AD6)
PA5 (AD5)
PA4 (AD4)
PA3 (AD3)
PA2 (AD2)
PA1 (AD1)
PA0 (AD0)
PB0 (T0)
PB1 (T1)
PB2 (AIN0)
PB3 (AIN1)
PB4 (SS)
PB5 (MOSI)
PB6 (MISO)
PB7 (SCK)
OC1B/PE2
PE0/ICP/INT2
GND
PC7 (A15)
PC6 (A14)
PC5 (A13)
PC4 (A12)
PC3 (A11)
PC2 (A10)
PC1 (A9)
PC0 (A8)
ALE/PE1
PD7 (RD)
PD6 (WR)
PD5 (OC1A)
PD4/XCK
PD3 (INT1)
PD2 (INT0)
PD1 (TxD)
PD0 (RxD)
U6
ATMEGA8515-TQFP
C1
22 uF
C2
100 nf
R3
22K
R9
10K
R6
10K
R7
10K
S1
A
COM
B
SWA
SW2
QUADRATURE SW
L1
330 µH
D1
1N4001
C16
22 uF
RL
LOAD
J4
Socket
R2
330
Q1
2N2222
Q2
2N7000
J5
CONN-H4

FIGURE 2
Example of a simple switching converter.

Inductor Saturation Current: The saturation current is the current at which the coil is saturated, where the inductance falls off, and the reactance goes very low. A switching supply depends on current in the coil changing. Once the current stops changing, the coil current is no longer a function of the reactance of the coil, but is instead limited by the resistance. The resistance will be significantly smaller than the reactance, so more current flows through the windings, producing more heat. Generally, you want to choose an inductor that never saturates in normal operation.

Inductor Resistance: The resistance is the ohmic resistance of the wire, itself. To demonstrate the effect of inductor resistance, I used two different inductors, one was a 330µH toroid on a ferrite drum core. The second inductor was a 330µH SMT inductor on a smaller drum core. The smaller core required smaller wire (more resistance) to achieve the same inductance. I walked through the frequencies from 12kHz to 25kHz, duty cycle 10% to 70%, and found that the maximum efficiency was 72% using the larger core, at 20kHz and 70% duty cycle. With the smaller core, the efficiency in the configuration was 62%. The larger core produced an output power of 959mW into the 500Ω load, and the smaller core produced 808mW.

I did the same experiment with two 250µH inductors, again using larger and smaller drum cores. This time, the maximum efficiency with the larger coil was 73% at 25kHz and 70% duty cycle. The smaller drum core produced 68% efficiency. Output power was 1,058mW and 1,049mW, respectively.

In a real regulator, you wouldn't be looking for the point of maximum efficiency or maximum power, but rather, holding a constant voltage or current. In this case, the parameters that gave a specific voltage for both inductors still had less efficiency with the smaller inductor. For example, the larger 330µH inductor had efficiency of 68% at 12V, and the smaller inductor was 66% when the duty cycle was set to produce 12V (both at 20kHz).

In both the 330µH and 250µH examples, you get better efficiency with the larger (lower resistance) inductor — but that also requires more PCB real estate.

OTHER EFFICIENCY FACTORS

Transistor Drive: Another factor that influences efficiency is the transistor drive, which affects the transistor turn-on and turn-off times. R2 limits the current into the base of Q1 to about 13mA. I placed another 330Ω resistor in series with R2, and the peak efficiency with the larger 250µH inductor went down 4%. The larger resistor increases the time needed both to drive Q1 into saturation, and to bring it out of saturation.

In a high-power switcher, having the drive transistor spend any significant time in the linear region will significantly increase heating and decrease efficiency. In an IC switching regulator with an integrated transistor, all of that is beyond your control. You just pick external components that are compatible with the IC. But in situations where you are using an IC with an external transistor, make sure the transistor is driven hard enough to keep efficiency up.

Bipolar Switching Transistor: An IC may have either an internal bipolar or MOSFET transistor. A bipolar transistor has a saturation voltage that limits the voltage across the inductor and is current-dependent. For example, in a boost switcher with a 5V supply voltage, if the transistor has a 0.5V saturation voltage, then the inductor will have 5 minus 0.5 or 4.5V across it. This limits voltage across the coil, and therefore, the current through it.

MOSFET Switching Transistor: A MOSFET transistor has an on-resistance measured in ohms. I swapped the 2N2222 for a 2N7000 MOSFET. The on-resistance resistance of the 2N7000 is typically 1.8Ω at room temperature and with a 4.5V gate drive voltage. This resistance is effectively in series with the inductor and has the same effect as adding that resistance to the inductor, itself. It also causes the transistor to dissipate some heat. Staying with the 250µH inductor at maximum efficiency, the 2N7000 produces 74% efficiency, close enough to the 2N2222 to be within the measurement error.

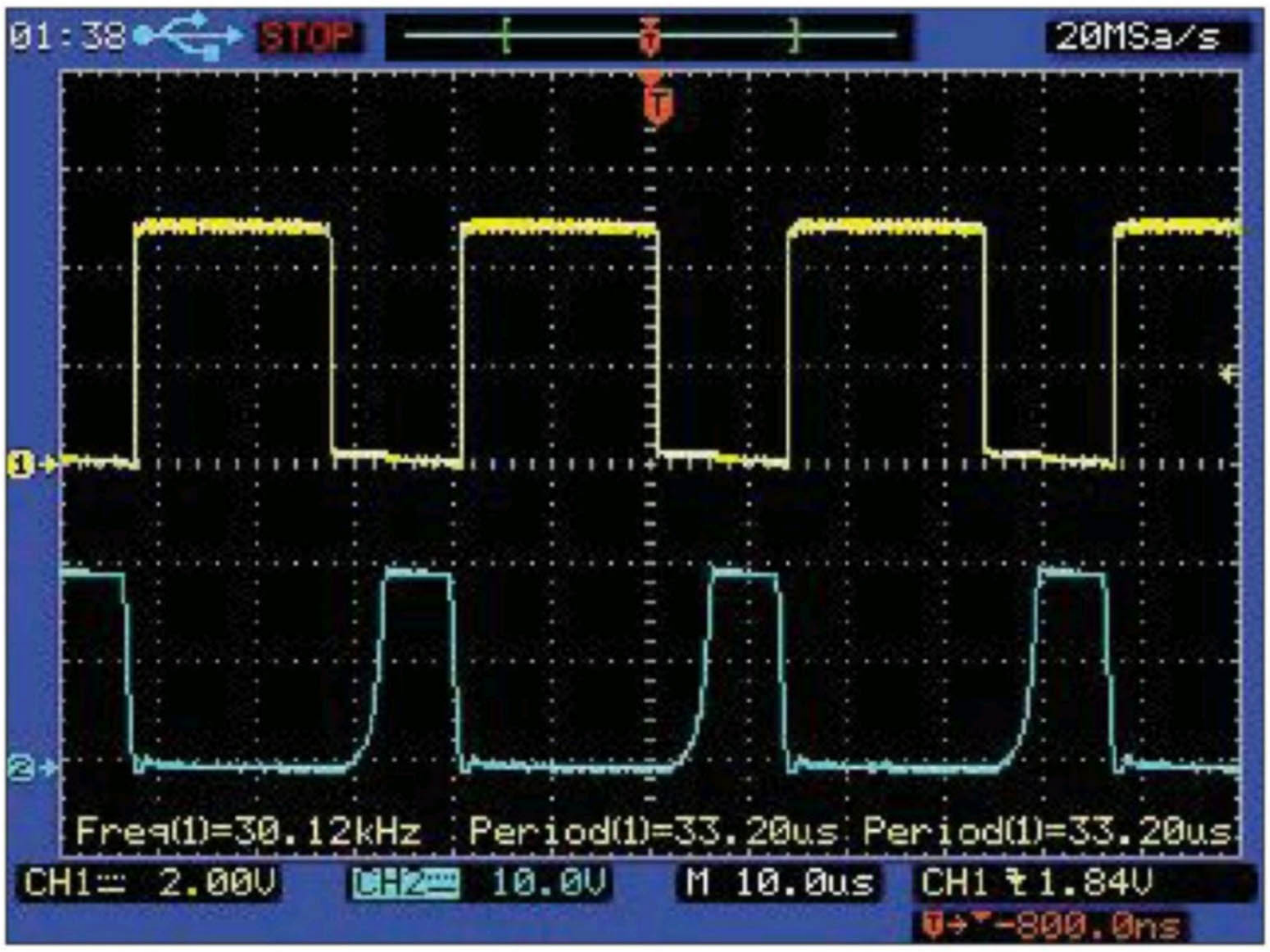


FIGURE 3
Effect of gate capacitance on power MOSFET response.

Power MOSFET: In a higher-power switcher, you'd use a power MOSFET with lower ON-resistance, instead of the 2N7000. To demonstrate the effect of this, I replaced the transistor with a power MOSFET with ON resistance under 0.02Ω. Staying with the maximum efficiency 220µH settings, the efficiency drops to 59%. Why? **Figure 3** shows what happens. The upper (yellow) trace is drive from the MCU, and the lower (blue) trace is the MOSFET drain. There is significant delay in switching the transistor, as well as a slow ramp when the transistor switches.

The reason is that the input capacitance is about 2,000pF for this transistor—40 times the capacitance of the 2N7000. So, yes, you will get better efficiency with a power MOSFET. Generally, however, the more current a MOSFET can handle, the more gate capacitance it has. So, to use a power MOSFET, you need something like 500mA of drive capability into the gate. You could add this to the example circuit, but it would require an IC driver or a few extra transistors.

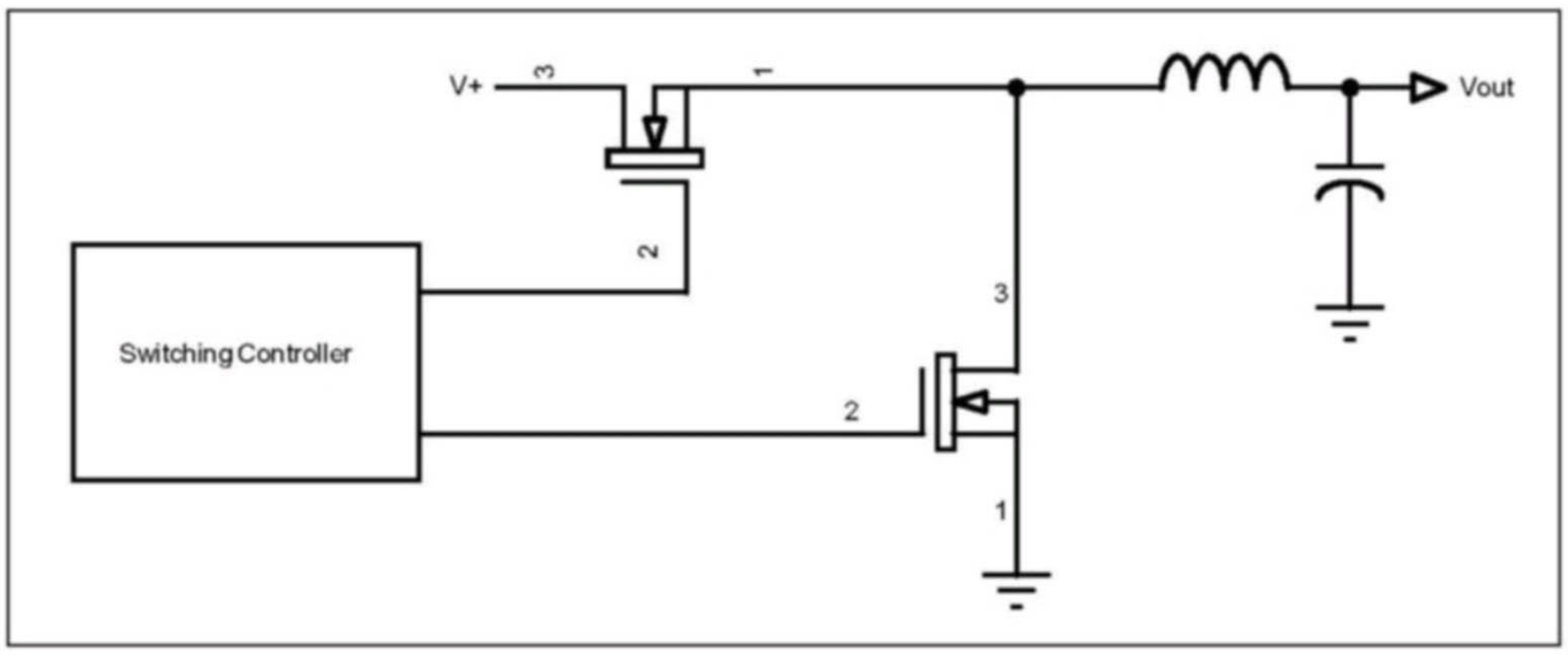


FIGURE 4
Synchronous rectifier.

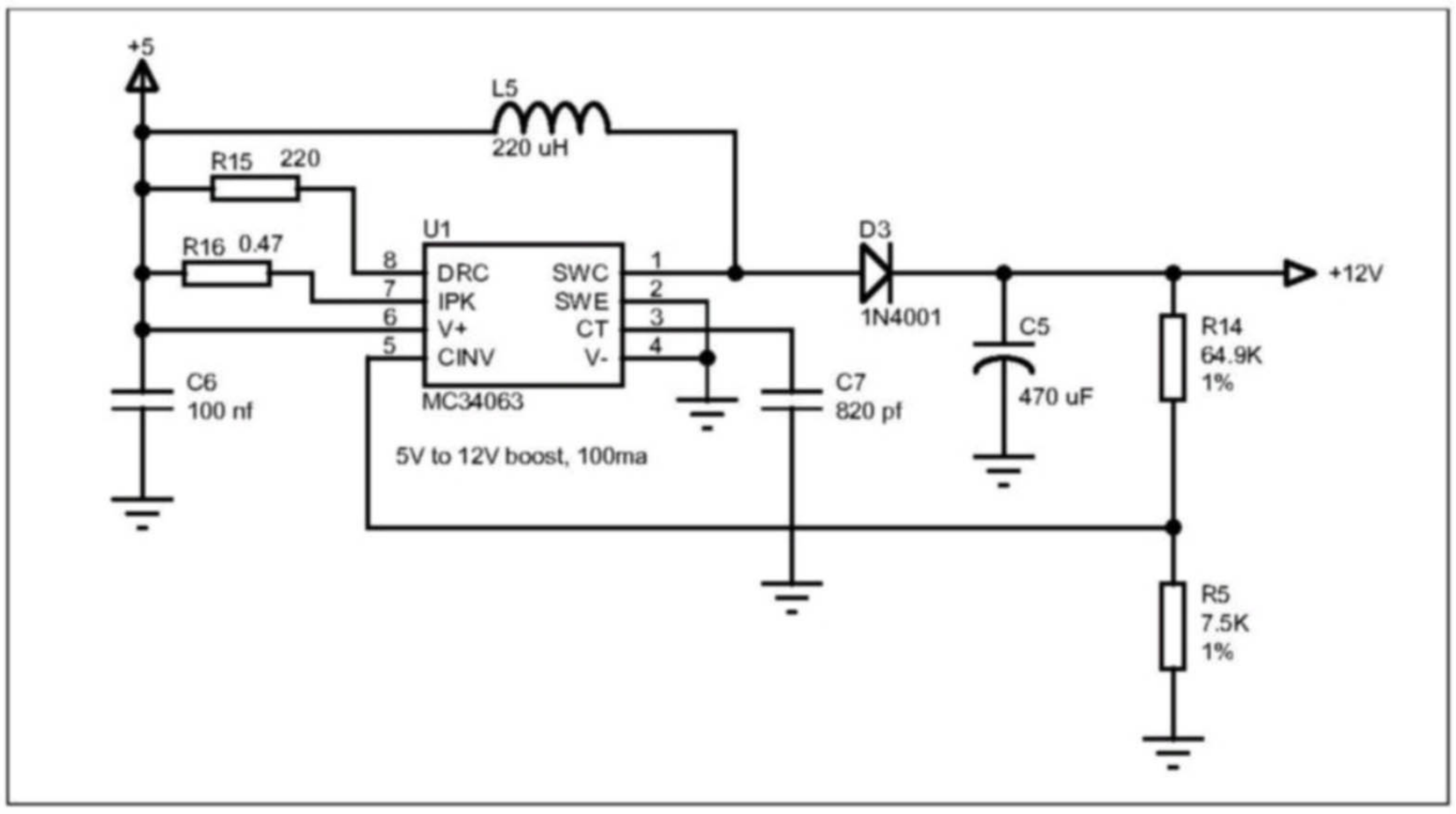


FIGURE 5
MC34063 example.

Additional materials from the author are available at:
www.circuitcellar.com/article-materials
References [1] to [3] as marked in the article can be found there.

External Transistor: Some switching regulator ICs have provision for an external transistor. In choosing such a transistor, and in general, a lower ON resistance (MOSFET) or lower saturation voltage (bipolar transistor) will maximize efficiency. Other components have to be compatible, of course, especially the aforementioned gate drive into a MOSFET.

EFFECTS OF DIODES AND OTHER RECTIFIERS

In the simple transistor example, I used an ordinary silicon rectifier diode. But in most switching supplies, you will use a Schottky rectifier. The 1N4001 has a forward voltage drop of about 800mV at 1A. The instantaneous current through the diode has to be higher than the switcher output current. If the diode is passing 1A during conduction, then it would use up 0.8W (800mW x 1A) while conducting. If we assume that the diode conducts 50% of the time (50% switcher duty cycle), then the average power consumption in the diode is 400mW.

Comparing the 1N4001 to a Schottky rectifier, a 1A Schottky diode might have a forward drop around 400mV, so it will waste only half the power of the 1N4001 at an output current of 1A. As with the ordinary rectifier diode, the forward drop of a Schottky diode goes up with forward current. So, to minimize losses in the rectifier, you will want a diode that has a small voltage drop at the expected output current.

The effect of the rectifier on efficiency depends on the output voltage, switcher duty cycle, and current. Say that the 1N4001 is used in a circuit with 1A conducting current and 50% duty cycle, as described. If used in a switcher supplying 12V, 500ma, the output power is 6W. The 800mV drop across a 1N4001 diode uses 800mW x 50%, or 6.6% of the output power. But if the switcher is a buck (step-down) converter with output of 3.3V at 500mA, then the power lost in the diode is 0.5 x 800mW/1.65W, or 23% of the output power—much more significant.

Synchronous Rectifier: The losses in the diode can be reduced by using a synchronous rectifier. A synchronous rectifier is a transistor that is turned on and off by the controller. Since a saturated transistor can have a much lower voltage than even a Schottky diode, the power lost in rectification can be reduced. This requires a controller that can time the switching transistor and the rectifier transistor. The TI LM5122 [2] is an example of a controller that drives external MOSFETS to perform synchronous rectification.

Figure 4 is a simple diagram of a synchronous rectifier in a step-down configuration. The controller turns on the top MOSFET to drive current into the inductor, and then turns on the lower MOSFET to ground the driven side of the inductor for the other part of the cycle. The controller has to control the MOSFETS so they are never on at the same time, to avoid shoot-through.

A REAL SWITCHER

Figure 5 shows a 5V-to-12V boost converter using the TI MC34063 switching regulator [2]. The MC34063 is an older part, and it lacks features of newer parts. It's useful for prototyping, because it is still available in a DIP package, the output voltage is programmable, and it can function as a step up-(boost) or step-down (buck) converter. It has internal switching transistors and can also drive an external transistor for higher current applications.

The circuit shown is configured to have about 100mA output current. The MC34063 datasheet [3] has the design equations for determining the various component values for the configuration you want. I won't go through those here. Resistors R5 and R14 determine the output voltage, and the regulator keeps the voltage at the feedback pin (pin 5) at 1.25V. With the R5/R14 values shown, that occurs when the output is 12.06V. The accuracy of the output value is dependent on the tolerance of the resistors, which is why I used 1% resistors in the example.

To evaluate the effect of the inductor on the efficiency of this circuit, I calculated (using the datasheet formulas) the optimum values for the circuit. The formulas will, of course, give you ideal values out to as many decimal places you want to calculate them, but you are limited to standard values for actual construction of the circuit, which is what the schematic shows.

The formulas calculate a minimum inductance and minimum output capacitance; for this circuit, I calculated 801pF for C7 and 183µH for L5. With the values shown in the schematic, I tried different inductors to demonstrate the effect on efficiency. The circuit was loaded with 250Ω, drawing about half the maximum current, so the output power was about 580mW. Input power with different inductors is shown in **Table 1**.

L5, µH	Input Current, mA	Input power, mw	Efficiency
500	186	930	62%
330	280	1.4	41%
200	160	0.8	72%
130	172	0.86	67%

TABLE 1
Input power with different inductors.

The circuit is not all that efficient with these component values; the MC34063 can approach 90% efficiency. But this shows the effect of the inductor value. The part was able to regulate the output voltage with nearly a 4:1 span of inductor values, but the efficiency is degraded as the inductor value moves up or down from the optimum value. Of course, the optimum value depends on the size of the load. In the circuit shown, with a 350Ω load (34% of maximum), the most efficient inductor was 200µH, and efficiency with that value was 72%. The point is that the inductor providing the optimum efficiency may vary with the output current and input voltage. If you are using a different switching regulator, you'll get different results.

A Practical Use**:** I used an MC34063 in the MCU switcher testing to allow measurement of input current. Any time you put a current meter in a DC circuit, there is a voltage drop across the meter. This voltage drop reduces the voltage to the circuit, so the MCU might only be getting 4.8V instead of 5V if the output power is 400mW. I constructed a 12V-to-5V buck converter to power the MCU circuit. The feedback that regulates the output voltage is connected to the MCU circuit after the DMM that is making the current measurement. A block diagram is given in **Figure 6**. This ensures that the voltage at the MCU is always 5V. The voltage at the DMM input might reach 6V or something, but the voltage to the MCU circuit will be constant, making the power measurements consistent.

CONCLUSION

Maximizing efficiency—and thereby minimizing wasted heat and energy—involves several factors and different components in the power supply design. I've tried to look here at the pieces that are involved and the effects they can have on that maximum efficiency goal. Even if you never need to design a power supply, understanding it is useful.

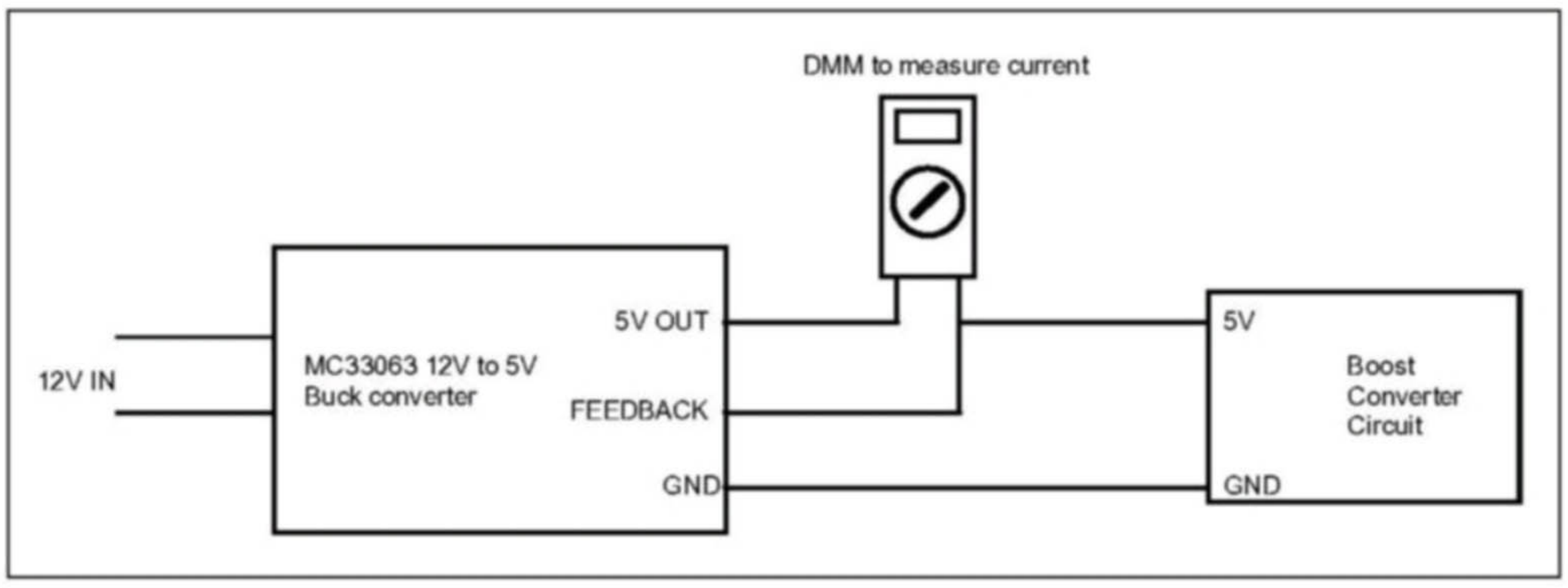


FIGURE 6
Current measurement.

ABOUT THE AUTHOR

Stuart Ball recently retired from a 40+ year career as an electrical engineer and engineering manager. His most recent position was as a Principal Engineer at Seagate Technologies.

Embedded in Thin Slices

Asynchronous Programming in Flutter

Shallow Dive into Google's Mobile App Framework

By
Bob Japenga

This month, Bob continues his look at mobile app programming from the perspective of an embedded system designer. He begins a shallow dive into Flutter, the mobile app framework from Google, by describing his transition from React Native to asynchronous programming in Flutter.

I introduced the Flutter ecosystem and discussed a little about Flutter in my article, "An Embedded Designer Looks at Mobile App Development," in the April 2024 issue of *Circuit Cellar* [1]. This month's article assumes you are familiar with the terms that were introduced previously, so it might be helpful for you to review my earlier column. Some definitions I will be using here are given in **Table 1**.

Being somewhat crazy, I took the entire app that I wrote in React Native, and rewrote it in Flutter. What an adventure! And it has taken me so much longer than the original React Native app. Overall, I like the Flutter structure better than React Native, but I find the non-orthogonality of the architecture and imprecise documentation completely maddening [2].

Programming in Flutter is like driving in Boston, where there are so many one-way streets that you are sure that some day you will be driving down a one-way street and reach an intersection with four streets all one-way to the spot you are in. "You can't get there from here!" Sometimes with Flutter, you find yourself unable to define the function you need with the types that you need, because of language restrictions.

Terms	Definition
Asynchronous operation	An operation that, once started, allows others to run concurrently
Asynchronous function	A function that contains at least one asynchronous operation
Future	A Dart class used in asynchronous programming indicative of a result to be provided sometime in the future.
future	An instance of a Dart Future class
Isolate	A concurrent thread in Flutter
Synchronous operation	An operation that runs to completion without interruption
Synchronous function	A function that only performs synchronous operations

TABLE 1
Some definitions used in this article.

FLUTTER/DART ASYNCHRONOUS PARADIGM: UNDERLYING ARCHITECTURE

We embedded systems designers need to know how things work under the hood, because without this information, we normally cannot understand the documentation provided. Here is my peek under the hood of Flutter asynchrony.

Flutter apps are run in separate memory spaces called "isolates." Think of a thread or task in other systems. In theory, all Flutter applications can run in one isolate, such as the "Hello World" example I gave in my April article. Any time you need to call an asynchronous operation, such as reading a URL from the Internet, Flutter will create a new isolate. Isolates cannot read or write each other's memory. They can send messages. You won't find any mutexes or semaphores or locking mechanisms like what we are used to.

Every isolate has an event loop that handles incoming events either synchronously or asynchronously. However, as a Flutter programmer, we don't have access to these internals probably for good reasons. But it creates real problems in creating real systems in our accustomed way.

I am still not sure if this paradigm is brilliant or 436 boxes of different kinds of spaghetti thrown into a pot. Let's dig into the basics (of course, in thin spaghetti slices).

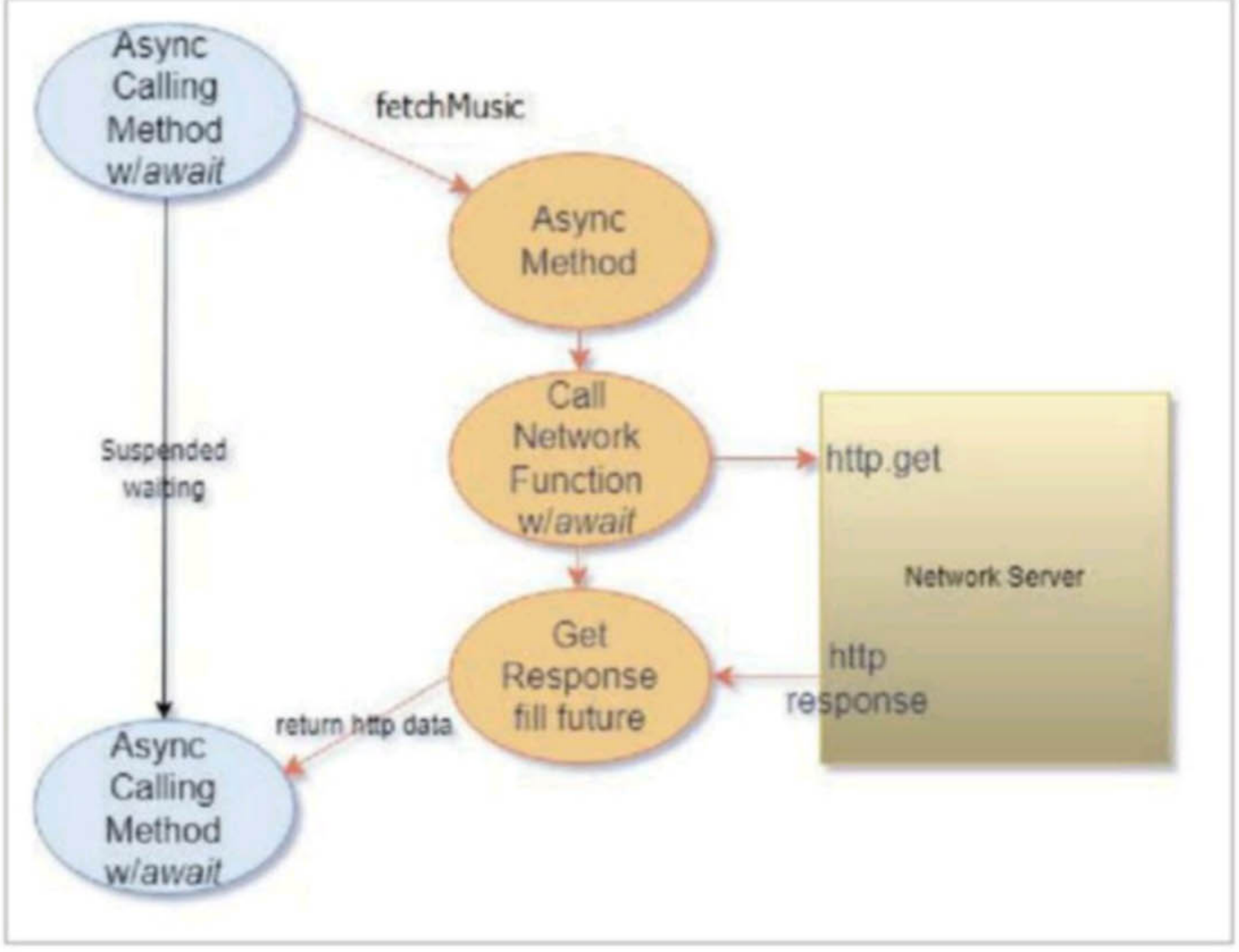

Async Calling Method w/await
fetchMusic
Async Method
Suspended waiting
Call Network Function w/await
http.get
Network Server
Get Response fill future
http response
return http data
Async Calling Method w/await

FIGURE 1
await Future.

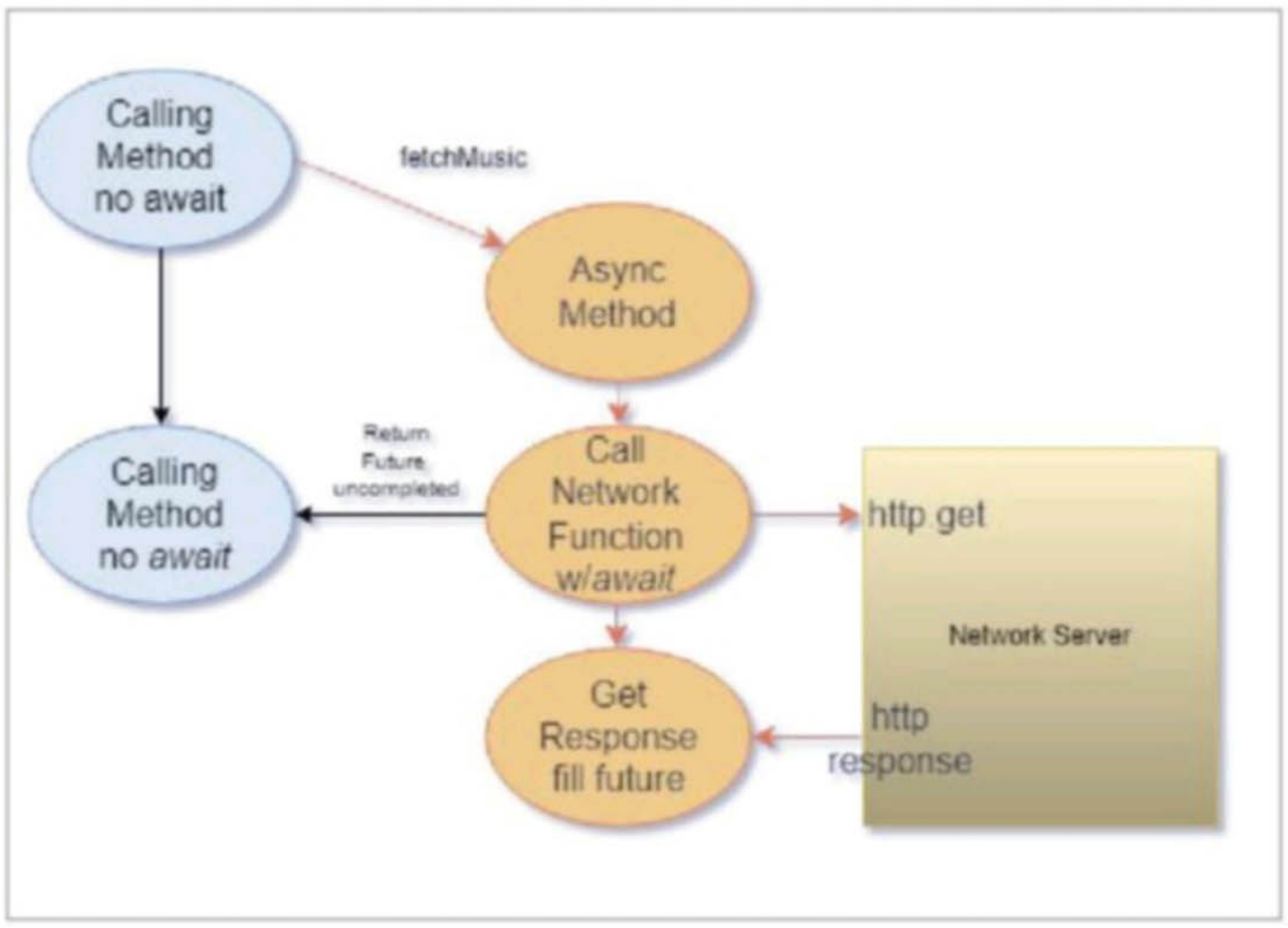

Calling Method no await
fetchMusic
Async Method
Return Future uncompleted
Calling Method no await
Call Network Function w/await
http get
Network Server
Get Response fill future
http response

FIGURE 2
No await Future.

PROGRAMMER'S PERSPECTIVE

The heart of Flutter's approach to asynchrony is the "Future class" (Table 1). The Future class works in a way similar to a Promise from JavaScript and React Native. It has two states: uncompleted and completed. The completed future will have either a value (on success) or an error (on failure). Any time you have an asynchronous action (for example, a function `foo` that calls asynchronous function `bar` that reads a URL from the Internet), `bar` returns a future. Sometimes it returns the future immediately, when `bar` doesn't wait for the response. In this case, the future is "uncompleted." If inside `bar`, it waits for the response, then it can return a "completed" future. However, `foo` can wait for `bar` or not. If `bar` waits for `foo`, it will get a completed future (**Figure 1**). If it doesn't wait, it will receive an "uncompleted" future (**Figure 2**).

The keyword in the language to "wait for a future" is the same as in JavaScript/React Native: `await`. Also, any function that wants to use await must be declared as an asynchronous function with the keyword: async.

As in React Native, the use of the `async` and `await` keywords is an attempt to make asynchronous operations look like synchronous operations, and thus easier to follow the flow. All the code flows linearly without bouncing around with callbacks and such.

Listing 1 is a code snippet of an asynchronous function that reads an mp3 file from the Internet.

We can see that the function `fetchMusic` is an asynchronous function that returns a future of type "Response" that is defined in a package called "http." It does not return immediately, but because of the `await` keyword, it returns after a successful or error response is received from the `http.get` function. Any function that calls `fetchMusic` will get an "uncompleted" future immediately, unless it also is an asynchronous function with an `await` before the call to `fetchMusic`. In my opinion, this feature creates a mess, because if you want synchronous-like, easy-to-follow flow, each function up the widget tree must be asynchronous, until one gets to the root function in a given isolate. (Note: "Uri.parse" simply formats the standard URL format into one that `http.get` handles). In a complex app like mine, you can accumulate more futures than the movie, Back to the Future.

There are lint-like tools that tell you when you have a discarded future (discarded_future), or one that you used before you waited for it (unawaited_future). Although that helps clean up the messes I make using futures, it is not the kind of problems I had using the standard, event-driven mechanisms that I am accustomed to.

```
Future<http.Response> fetchMusic () async {
  return await http.get (Uri.parse(‘https://MUSIC.com/albums/1’));
}
```

LISTING 1
This Flutter code snippet uses async and await.

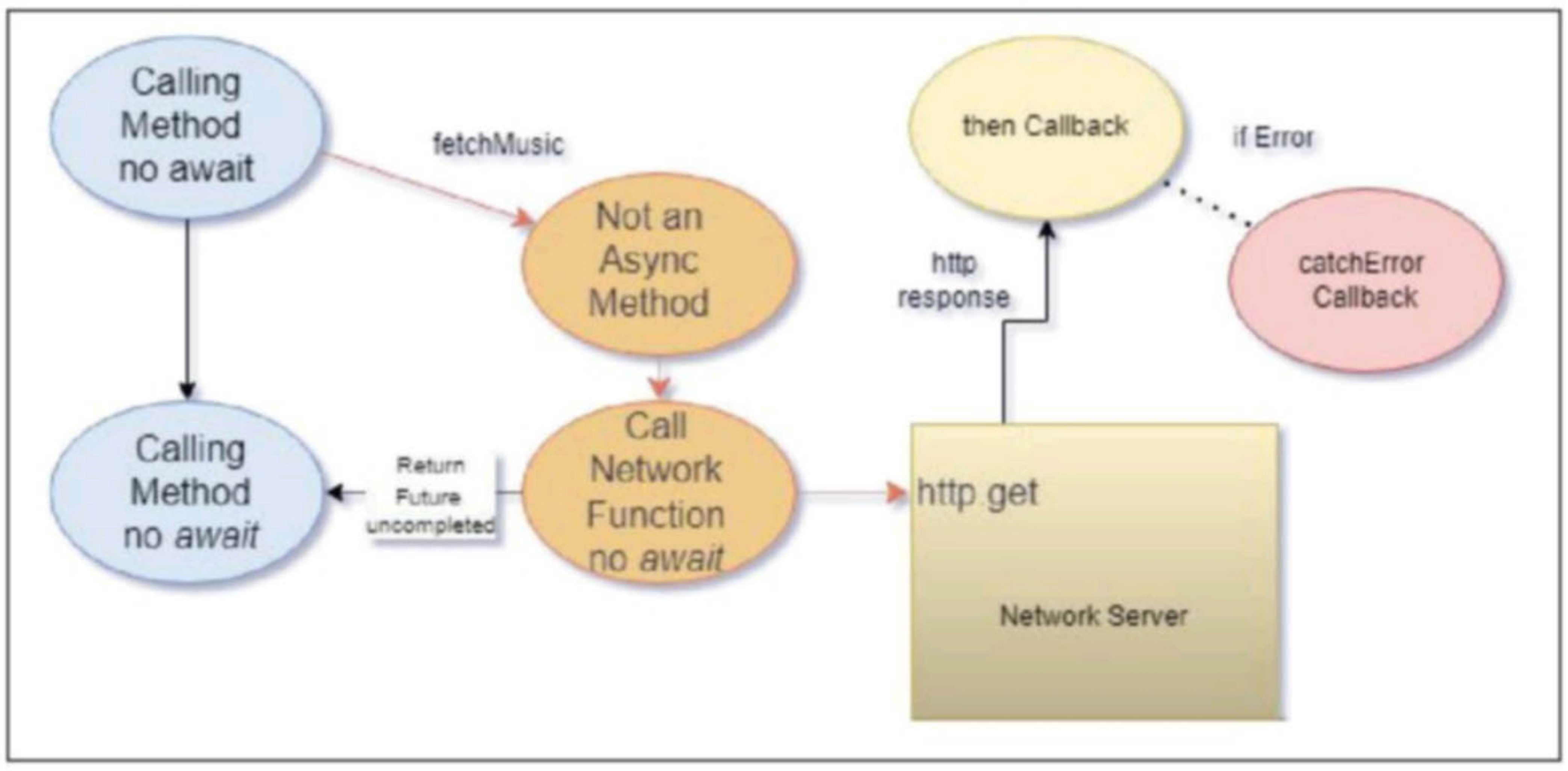


FIGURE 3
Callback method.

```
void fetchMusic () {
    http
        .get(Uri.parse(‘https://MUSIC.com/albums/1’))
        .then(callbackFunctionForCompleted)
        .catchError(callbackFunctionForError);
  }
```

LISTING 2
This Flutter code snippet uses asynchronous callbacks.

Paradigm	Description
async	The method will spawn a new isolate and returns a Future
Future	A Dart Class that can have the following states: uncompleted, completed with success or completed with errors
FutureBuilder	A widget that allows computations based on the state of a future
StreamBuilder	A widget that allows computations based on the state of a Stream of futures.
then()	Registers a callback function for an async function called upon completion of a future
delay()	Sets a future after a delay
whenComplete()	Registers a callback function for an async function upon completion of a future
catchError()	A callback for an async function upon completion of a future with an error. Usually invoked on the result of a then() function.
timeout()	A callback invoked after a specified period of time. For example if the error response from an Internet fetch is too long, you can use the timeout callback to respond more quickly.
any()	A function that returns the completed result of the first future completed in a list.
doWhile()	Performs a computation until a future Boolean operation is false
forEach()	Performs a computation for each item in an iterable list. Returns a future that completes with null when all elements have been processed.

TABLE 2
Summary of several other operations that support asynchronous programming.

CALLBACKS

Any time you pass a function name (`foo`) to another function (`bar`), `foo` is a callback function. The name is misleading, because the function—`foo`—doesn't necessarily call back function `bar`. Flutter also allows you to use callbacks to deal with asynchronous actions. Again, following JavaScript/React Native, the language has built-in callbacks such as `then` and `catchError` and `whenComplete`. Let's look at the same `fetchMusic` function with callbacks (**Figure 3**).

Note a couple of differences. `fetchMusic` does not need to be asynchronous. It does not need to return a Future. The response is passed to the "callback for completed" function, which in turn can call the "callback for error" function (**Listing 2**). The advantage of the callback method is that it strips away any illusion of writing synchronous code. You know that you are bouncing around.

Another way that callbacks are used in asynchronous programming is with the `wait` callback, which can be called when one or more futures are complete. Imagine you are reading several pieces of data off the Internet, and a subsequent screen needs to display all of this data. You can use the `wait` callback to create one future that will be complete when all are complete. In the example code later in this article, you will see this used with two futures.

There are a few other callbacks to handle various conditions. The `timeout` method (if the asynchronous call doesn't return in a reasonable time) and `whenComplete` (which is used like `wait` but for only a single future) are two such examples.

FUTURES ON SCREEN

Regardless of the way we handle futures (with `async/await` or callbacks), you still need a way to build screens with data from futures. Let's say that you read-in a bunch of data when the first screen is displayed. But on the second screen you display the data. How do you hold off the second screen while downloading the data?

For this, they have created a widget called "FutureBuilder." It allows you to build some screen widgets with the future's data when it is completed, and to build other screen widgets while waiting for the future to complete (for example, a spinning circle). The classic example is when we read data over the Internet. The response from the `http.get` is a future. The FutureBuilder allows you to: 1) display a widget (or widgets) indicating "waiting" when the future is incomplete; 2) display errors when the future is complete but with errors; and 3) display the http data when the future is complete and valid.

If you have a stream of futures (think of a stream of characters), another widget called "StreamBuilder" can be used. I will leave that to you to research if you want.

Several other operations that support asynchronous programming are summarized in **Table 2**.

EXAMPLE CODE

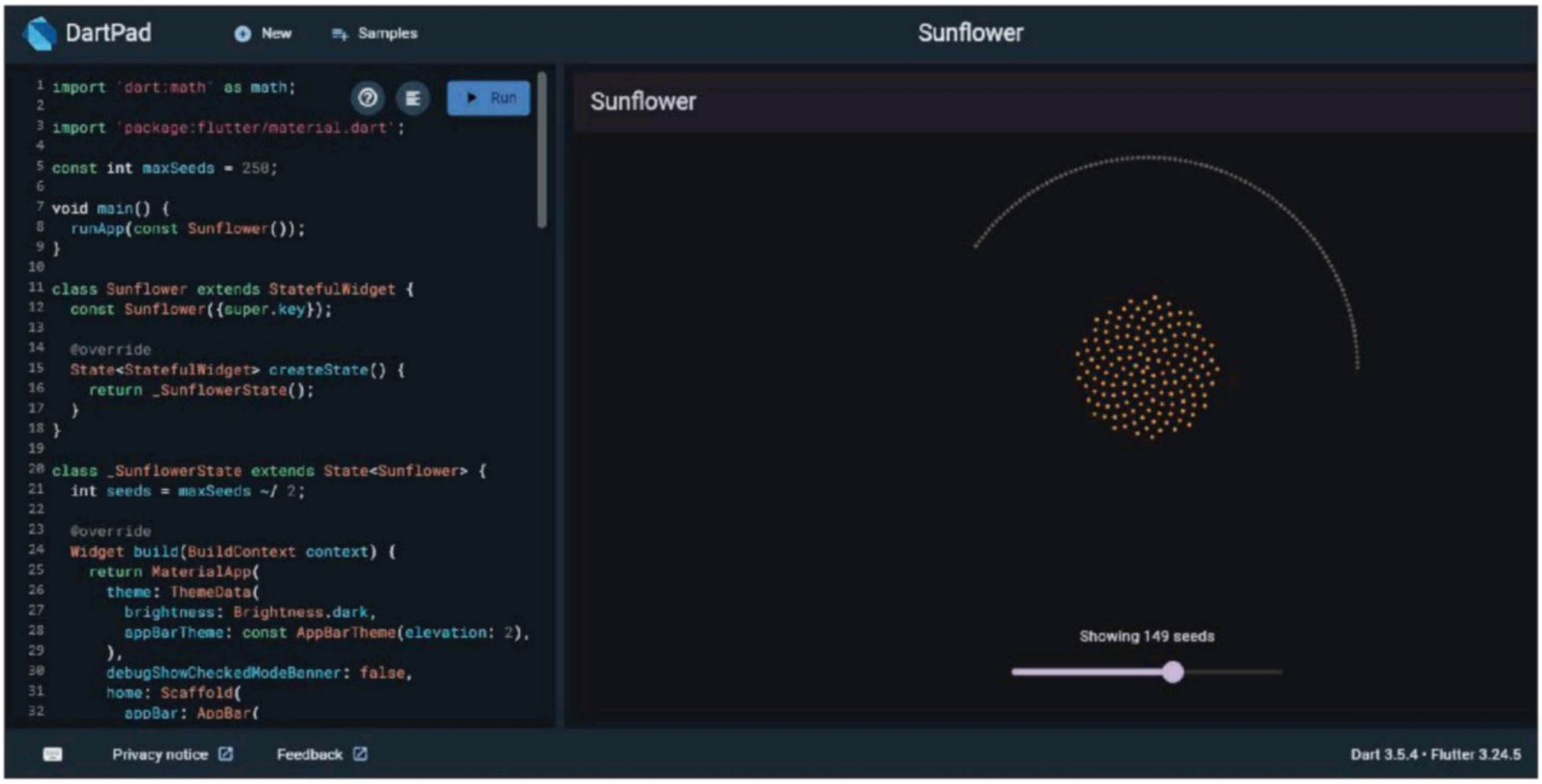
Flutter apps are written in a language called Dart. A Flutter app taken from the DartPad website [3] displays a sunflower with a variable number of seeds. I have modified it to demonstrate some asynchronous programming in Flutter. I annotated that code to walk you through all the details [4]. A screenshot of part of this DartPad webpage with the code and sunflower seeds is shown in **Figure 4**.

In the rest of this article, I want to show you the asynchronous programming aspects of this app. But before we start looking at code snippets, let me tell you how you can run this. Unfortunately, in Flutter there is nothing like React Native Snacks, which I introduced in my April 2024 column [1]. DartPad only works as a Web app. You cannot use it on a simulator or a real phone, like you can with Snacks. However, you can run some very simple apps in DartPad in Web mode. For our purposes, if you want to try my example code, cut and paste it from the references into DartPad and run it.

'await' **Future**

The Sunflower app displays some text on the screen. Think of it as the default data. Then, after some delays, new text is displayed, indicating that a future has completed. One of the delays uses the `await` paradigm I spoke of earlier.

Notice that our method `awaitFuture` is an `async` method (**Listing 3**). I am using an artificial delay to simulate reading something



```
import 'dart:math' as math;

import 'package:flutter/material.dart';

const int maxSeeds = 250;

void main() {
  runApp(const Sunflower());
}

class Sunflower extends StatefulWidget {
  const Sunflower({super.key});

  @override
  State<StatefulWidget> createState() {
    return _SunflowerState();
  }
}

class _SunflowerState extends State<Sunflower> {
  int seeds = maxSeeds ~/ 2;

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      theme: ThemeData(
        brightness: Brightness.dark,
        appBarTheme: const AppBarTheme(elevation: 2),
      ),
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
```

FIGURE 4
Screenshot of part of the DartPad screen containing the example code.

```
Future<bool> awaitFuture () async {
    try {
      // Think of this like reading data from a slow internet
      await Future.delayed(const Duration (seconds: 5));
      setState (() {
        awaitFutureStatus = 'awaitFuture 5 second wait is Complete';
      });
    } catch (e) {
      debugPrint ("Error $e in the Delay");
    }
    return true;
  }
```

LISTING 3
The Flutter code snippet uses the Future.delayed callback.

```
bool thenFuture () {
  Future.delayed (const Duration(seconds: 10))
    .then (successCallback)
    .catchError (failureCallback);
  setState (() {
   thenCallbackStatus = ‘thenFuture is waiting 10 seconds for callBack’;
  });
  return true;
 }

 FutureOr<dynamic> successCallback (dynamic) {
  setState (() {
   thenCallbackStatus = ‘then Callback is Now Complete’;
  });
  return true;
 }

 FutureOr<dynamic> failureCallback (error) {
  debugPrint (“Error in the Delay”);

  setState (() {
   thenCallbackStatus = ‘Future has now failed with $error’;
  });
  return true;
 }
```

LISTING 4
This Flutter code snippet uses a synchronous function with callbacks.

from the Internet that takes 5 seconds. The simulated fetch is wrapped in a try / catch. Remember that if the calling program does not wait for `awaitFuture` to complete, it will keep on going with the return value as a Future Boolean. The `debugPrint` function prints to the console only in debug mode. The call to `setState` tells the render engine to rebuild the screen with the “completed” status string (in this case, `awaitFutureStatus`).

‘then’ **Future Callback**

Alternately, the code in **Listing 4** illustrates the Future – `then` – `catchError` paradigm.

Note that `thenFuture` is not an `async` function. When the Future is completed, either the success or failure callback will be called. Meanwhile, `thenFuture` will immediately return true without waiting, and the calling function will continue on its merry way. The callbacks can do meaningful work and fill some future data for use at a later time.

Two additional Future Callbacks, `wait` and `whenComplete`,are also illustrated in the sample code (See the code included in the additional materials).

CONCLUSION

Well, there you have it—a shallow dive into the world of asynchronous programming in Flutter. As with React Native, the paradigm is so different from what I was familiar with as an embedded programmer, it gave me much grief. The documentation [4] [5] [6] didn’t help either. I hope that my attempt to clarify things has not muddied the water. Use the sample code and play around with the various methods illustrated.

In my next article, I will look at the other bugaboo for me with Flutter: state management. Wait! Didn’t I have problem with state management with React Native? Yes. And that is why there are add-ons to both frameworks to attempt to set the matter straight. I will try to create state management with what Flutter and Dart provided. But of course, only in thin slices.

ABOUT THE AUTHOR

Bob Japenga has designed embedded systems since 1973. From 1988-2020, Bob led MicroTools, an engineering firm specializing in creating a variety of real-time embedded systems across a wide range of industries. Bob has been awarded 11 patents in many areas of embedded systems and motion control. Now retired, he enjoys building electronic projects with his grandchildren. You can reach him at Bob@ListeningToGod.org.

Additional materials from the author are available at:
www.circuitcellar.com/article-materials
References [1] to [6] as marked in the article can be found there.

By Kirsten Campbell **PRODUCT NEWS**

Infineon and Flex Showcase Zone Controller Design Platform for Software-Defined Vehicles

Infineon Technologies AG and Flex, a diversified global manufacturer and new preferred automotive design partner for Infineon, are showcasing the new Flex Modular Zone Controller design platform for software-defined vehicles at CES 2025. The platform is an innovative, scalable zone control unit (ZCU) that features a modular microcontroller (MCU) architecture and common hardware building blocks.

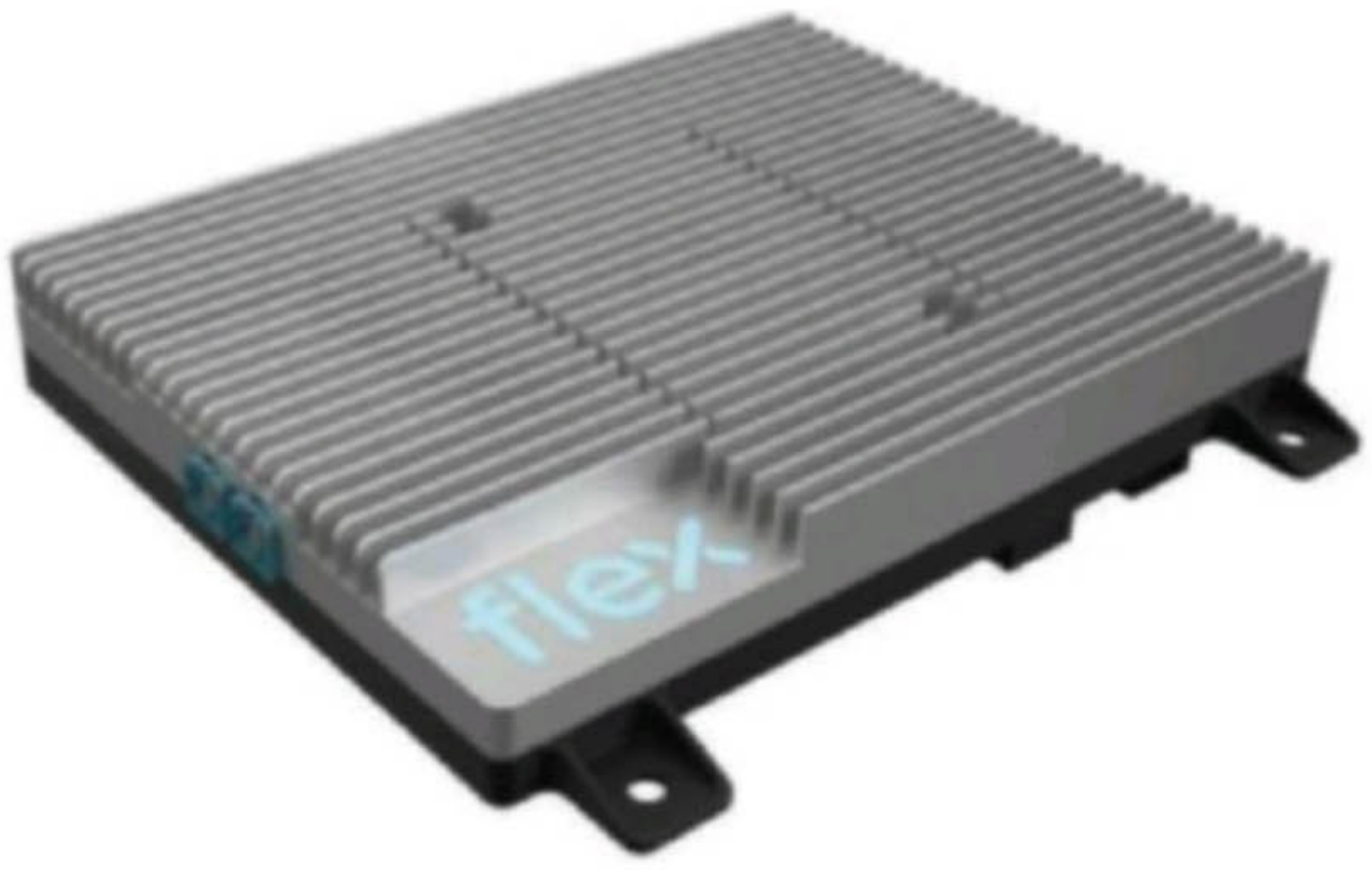
The Flex Modular Zone Controller platform is an automotive-grade design solution for rapid ZCU implementation, enabling automakers to deliver software-defined vehicles at scale with resilience. The combination of Infineon's innovative chipsets with Flex's design and advanced manufacturing capabilities provides automakers a next-generation zone controller platform with optimized Power-Distribution, Gateway and Motor-Control solutions. This production-ready platform can be quickly adapted to automakers' specific requirements, speeding manufacturing ramp-up and time-to-market.

"We at Infineon are very pleased to be working with Flex, whose design expertise complements our strengths," said Hans Adlkofer, SVP Automotive Systems Group of Infineon. "The combination of our achievements in cost and performance-optimized chipsets with the flexibility and the "design house-typical" development speed enables our customers to adapt their ZCUs to their specific requirements."

"As software-defined vehicles pave the way for next-generation mobility, the automotive industry must embrace partnerships that empower automakers to move faster and focus their resources on creating brand-defining experiences through software," said Mike Thoeny, President, Automotive, Flex. "Flex's collaboration with Infineon on advanced compute platforms exemplifies how deep ecosystem partnerships can enable automakers to launch software-defined vehicles with greater resilience and speed."

The Flex Modular Zone controller platform optimizes ZCU development by enabling efficient and fast implementation. Its optimized chipset from Infineon supports rapid evaluation of key component features, including Hardware Data Accelerators (Date Routing Engine/CAN Routing Engine), sequential diagnostics, I²T protection and SPI control, enabling early performance optimization. With endpoint-agnostic control implementation, the platform allows seamless integration of the modular MCU with power distribution and actuation functions.

Infineon | infineon.com

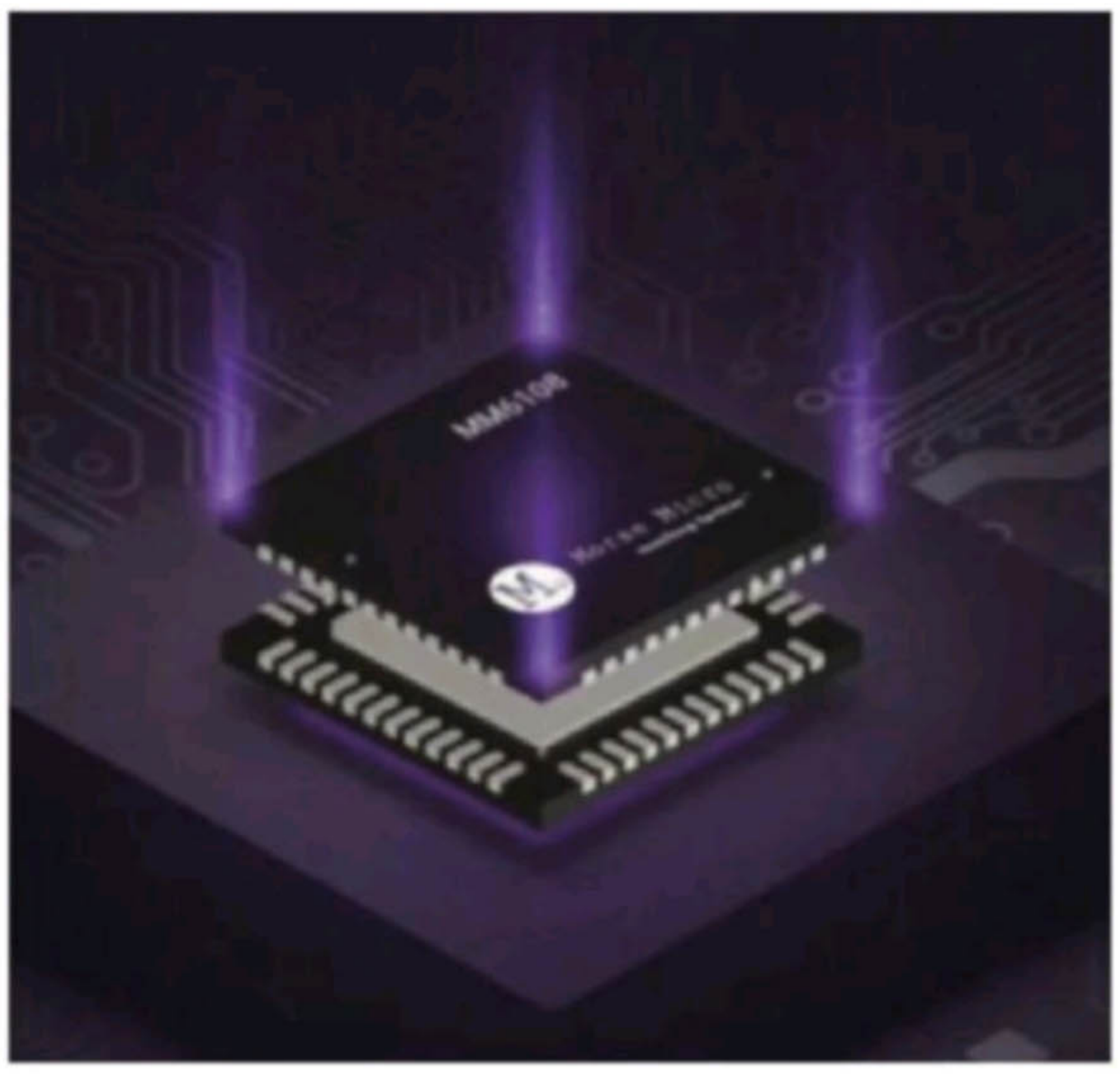


Morse Micro Launches Highly Anticipated Second-Generation MM8108 SoC

Enabling improved spectrum efficiency to minimize interference in high-density environments, the MM8108 delivers class-leading data rates of up to 43.33Mbps using world-first sub-GHz 256-QAM modulation at an 8MHz bandwidth, making it ideal for a range of applications in agricultural, mining, industrial, home, and city environments. Its integrated 26dBm power amplifier with high efficiency, and low-noise amplifier (LNA) ensure exceptional performance and enable global regulatory certification without the need for external Surface Acoustic Wave (SAW) filters. The exceptional power efficiency significantly extends battery life and enables the uptake of solar-powered Wi-Fi HaLow connected cameras and IoT devices.

The MM8108 Wi-Fi HaLow SoC offers enhanced security that supports next-generation WPA3 with Simultaneous Authentication of Equals (SAE) and GCMP encryption for robust link-layer protection. Also, USB, SDIO, and SPI host integration enables USB AP and STA dongle operation on new and existing network infrastructure. The compact 5mm x 5mm BGA package minimizes PCB size and cost.

The MM8108 is optimized for battery-operated applications with significantly extended sleep times and ultra-low power consumption in sleep modes. With class-leading Rx and Tx performance, it provides the fastest, longest reach Wi-Fi HaLow chip on the market, enabling countless applications including AI-enabled IoT devices and streaming of multiple 4K cameras in real-time over an extensive range.



Complementing the cutting-edge MM8108, Morse Micro is also unveiling the MM8108-RD09 USB dongle reference design, which demonstrates the ease of upgrading new and existing Wi-Fi 4/5/6/6E/7 network infrastructure to support Wi-Fi HaLow. The dongle reference design will be made available to customers as a fully compliant Wi-Fi HaLow reference design, which can be used to build commercial USB dongles or to simplify the integration of the MM8108 silicon in end products.

The MM8108-RD09 USB dongle will be bundled with Raspberry Pi 4B, power supply, and antenna, and made available as a complete Wi-Fi HaLow evaluation kit (the MM8108-EKH19) targeting customers interested in testing and integrating the MM8108 within their platforms. This strong combination puts the MM8108 at the center of a ready-to-deploy, compact solution that robustly integrates Wi-Fi HaLow connectivity into existing devices with exceptional ease and efficiency.

Morse Micro | morsemicro.com

NEW PRODUCT SUBMISSIONS— E-mail: product-editor@circuitcellar.com

TEST YOUR EQ

Contributed by David Tweed

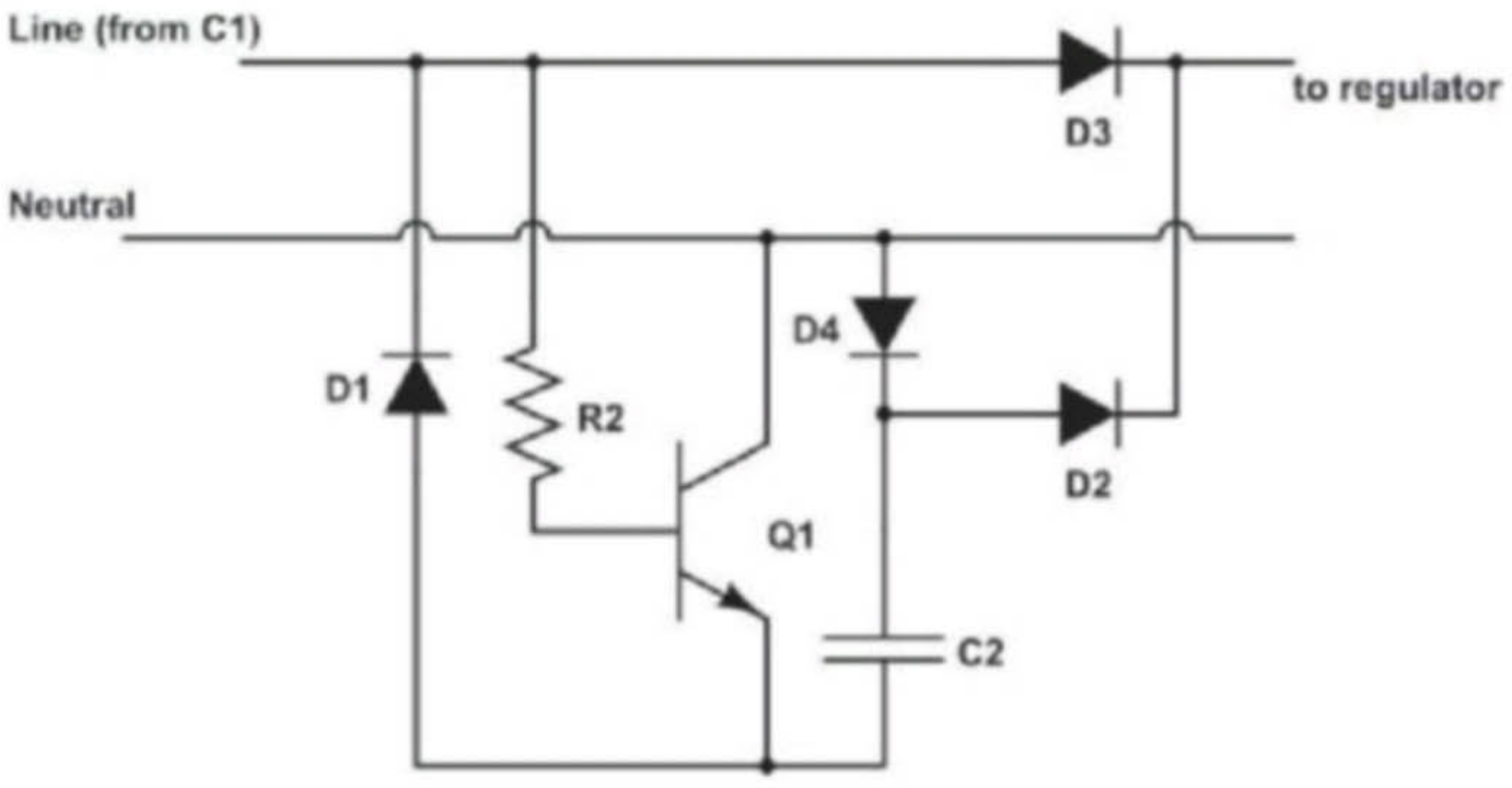
Answer 1 —

In general, and all other things being more-or-less equal, I2R losses are going to be the chief cost of efficiency in a switching power converter. Therefore, it would be a goal to keep currents as low as possible.

This would imply that the overall system efficiency will be better with a buck converter than with a boost converter. The average currents in the batteries, coil and the switching element will all be lower.

Answer 2 —

The transistor and C2 implement a "charge pump" that increases the available output current. It's easier to see if you redraw the central part of the circuit:


Line (from C1)
to regulator
D3
Neutral
D4
D1
R2
D2
Q1
C2

When "Line" goes negative, C2 charges through D1 and D4. Q1 is cut off, as are D2 and D3.

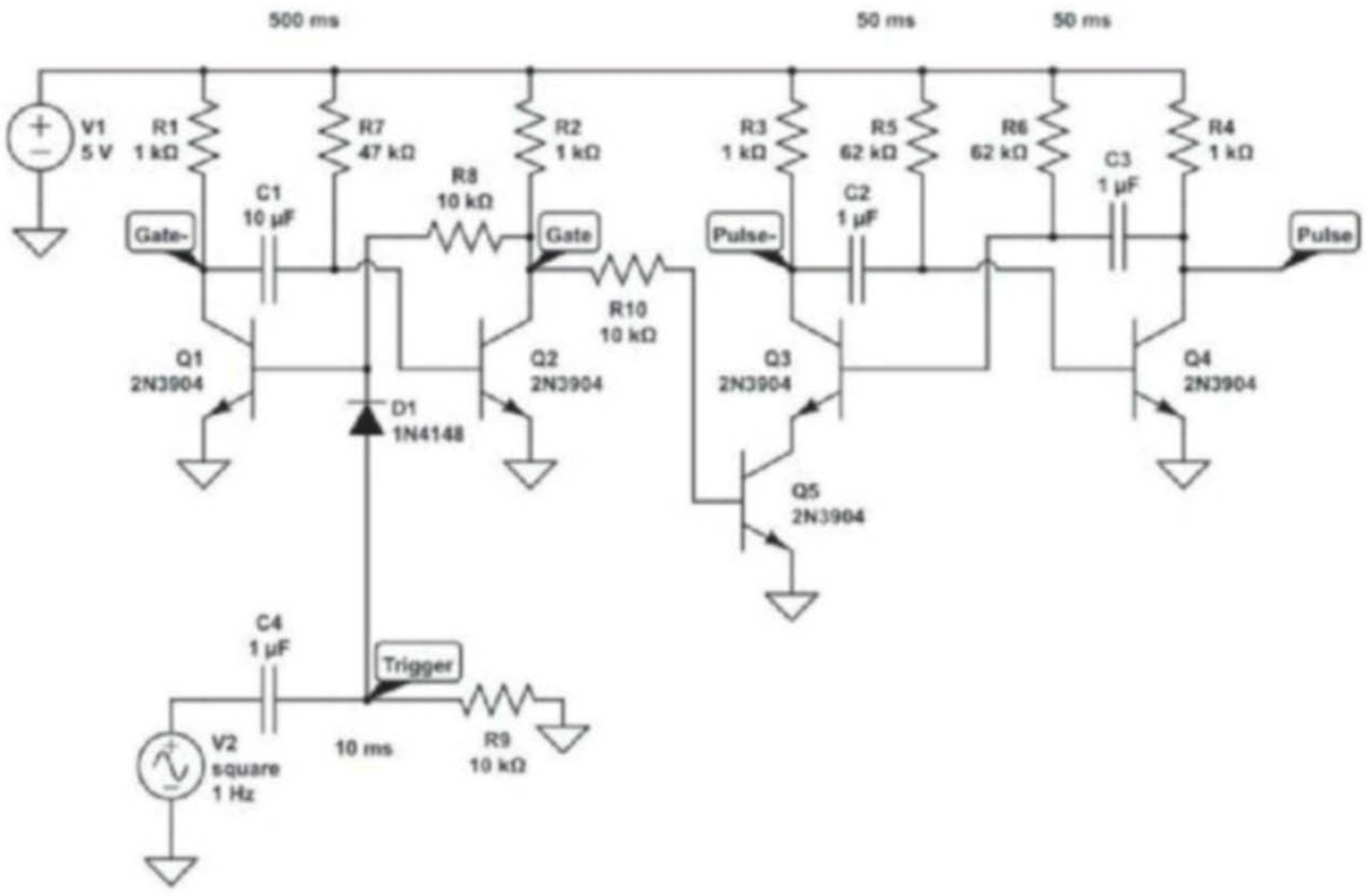
When "Line" goes positive, it not only supplies current through D3 to the load, it also switches on Q1. This pulls the negative end of C2 up to the "Neutral" voltage, allowing it to discharge through D2 and supply additional current to the load.

Answer 3 —

The same increase in available current could have been achieved with an ordinary 4-diode bridge rectifier. However, this circuit has the advantage that the negative side of the output (to the regulator) remains directly tied to neutral side of the input.

Answer 4 —

Use a monostable multivibrator to "gate" an astable multivibrator. It requires just 5 NPN transistors. With the values shown, it generates 4 pulses at 10 Hz each time a rising edge occurs on the trigger signal. Adjust R5 and R6 for the frequency and duty cycle of the pulses. Adjust R7 for the number of pulses.


500 ms
50 ms
50 ms
V1
5 V
R1
1 kΩ
R7
47 kΩ
R2
1 kΩ
R3
1 kΩ
R5
62 kΩ
R6
62 kΩ
R4
1 kΩ
C1
10 µF
R8
10 kΩ
C2
1 µF
C3
1 µF
Gate-
Gate
Pulse-
Pulse
R10
10 kΩ
Q1
2N3904
Q2
2N3904
Q3
2N3904
Q4
2N3904
Q5
2N3904
D1
1N4148
C4
1 µF
Trigger
V2
square
1 Hz
10 ms
R9
10 kΩ

HIGH PERFORMANCE, HIGHLY SCALABLE 30V-75V PORTFOLIO OF BUCK CONVERTERS

Optimized to power the highest precision loads with excellent accuracy

- Adjustable V_{OUT} from 0.8V to ~90% of V_{IN}
- ± 1% FB voltage accuracy over -40°C to 125°C
- Default 400kHz or adjustable switching frequency from 200kHz to 800kHz
- Available in 20ld QFN and 16ld HTSSOP packages

Visit **renesas.com/buck-integrated-fets** to learn more

RENESAS

IDEA BOX

The Directory of PRODUCTS & SERVICES

AD FORMAT:
Advertisers must furnish digital files that meet our specifications (www.circuitcellar.com/mediakit).

All text and other elements MUST fit within a 2" x 3" format.
E-mail adcopy@circuitcellar.com with your file.

For current rates, deadlines, and more information contact
Hugh Heinsohn at 757-525-3677 or Hugh@circuitcellar.com

When it comes to robotics, the future is now!

Advanced Control Robotics simplifies the theory and best practices of advanced robot technologies, making it ideal reading for beginners and experts alike.

With this book, you'll learn about:
- *Communication*
- *Technologies*
- *Control Robotics*
- *Embedded Technology*
- *Programming Language*
- *Visual Debugging...*

and more

Get it today, cc-webshop.com

ADVANCED CONTROL ROBOTICS
HANNO SANDER
start
start task async
repeat
start task sync and wait
task async
repeat
print text
wait 1000
task sync
print text
wait 1000
HANNO SANDER
circuit cellar

RAPID18

Universal Development Kit for PIC18 Family

ONLY $99!

- Real-time Clock/Calendar with Supercap
- USB port for text communication to the running program
- Built-in bootloader for loading code
- Excellent for AC/DC Data Logging

Kit includes everything you need to develop with a PIC® MCU:
- Rapid 18 Prototyping Board
- Single-Chip IDE C Compiler
- Exercise Book
- Power Adapters and Cables

sales@ccsinfo.com (262) 522-6500 x35
www.ccsinfo.com/cc1223

PIC® MCU and dsPIC® are registered trademarks of Microchip Technology Inc.

DEVELOPMENT KIT

With the right tools
designing a microprocessor can be easy.

Okay, maybe not easy, but certainly less complicated. Monte Dalrymple has taken his years of experience designing embedded architecture and microprocessors and compiled his knowledge into one comprehensive guide to processor design in the real world.

Monte demonstrates how Verilog hardware description language (HDL) enables you to depict, simulate, and synthesize an electronic design so you can reduce your workload and increase productivity.

cc-webshop.com

Verilog HDL
Microprocessor Design Using Verilog HDL
by Monte Dalrymple
CIRCUIT

www.embeddedTS.com

TS-7100

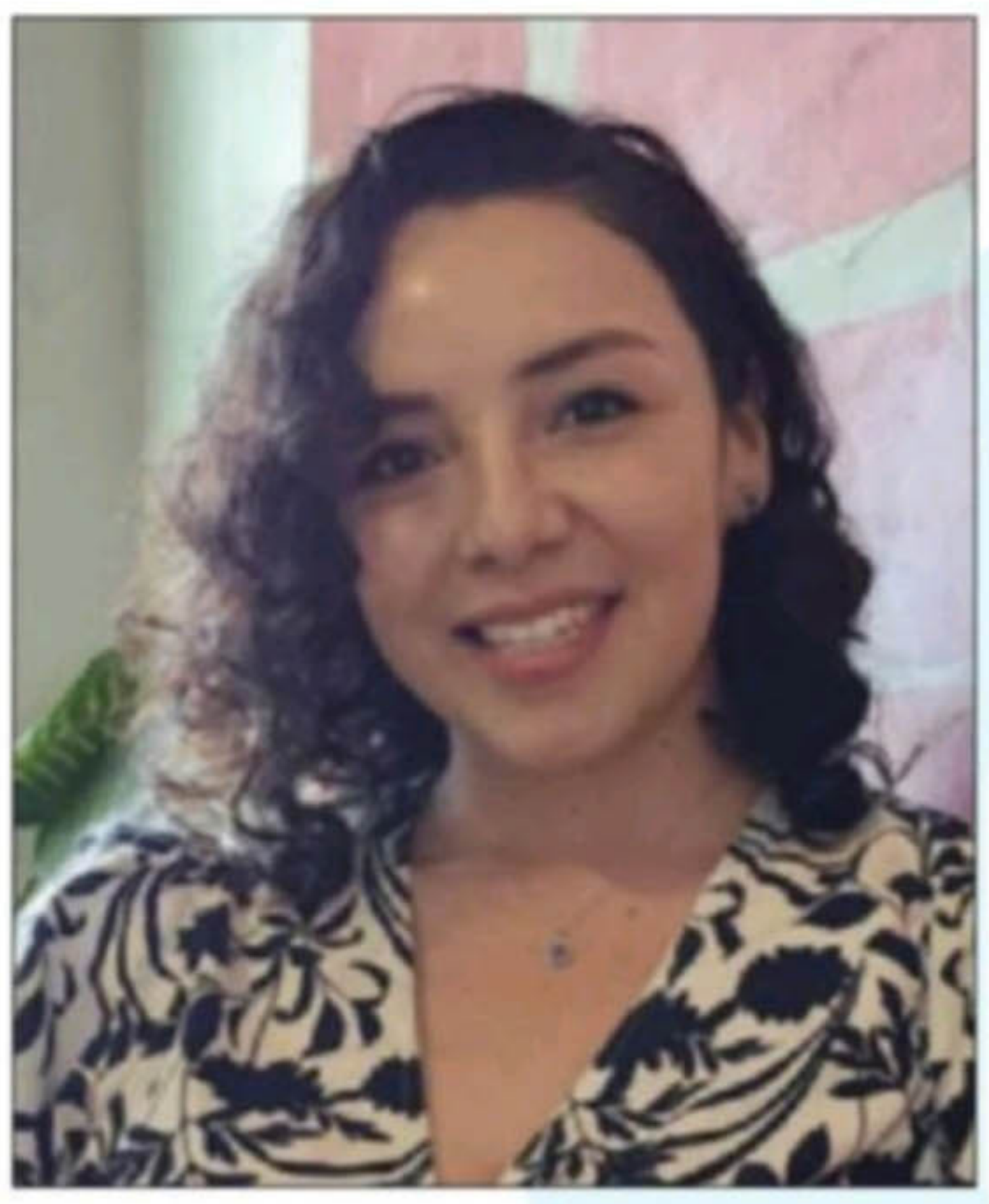
- NXP i.MX 6UL Arm Cortex®-A7 @ 696 MHz
- 512 MB RAM
- 4 GB eMMC Flash Storage
- Ethernet, USB, RS-232, RS-485, and CAN
- Dual Ethernet, WiFi, Bluetooth
- NimbeLink/Digi socket

embeddedTS

TS-7100
ANALOG
CAN
DIG IN
RS-232
GROUND
RELAY 2
RELAY 1

The Future of Development Boards

NXP MCX MCUs and IDEs Cut Development Time



Development boards, such as NXP's FRDM platform for MCX A series MCUs, are revolutionizing embedded systems by integrating components for prototyping and innovation in the IoT landscape. These technologies streamline development, enhance flexibility, and reduce costs, enabling greater focus on software development while addressing complex challenges.

By
Mónica Cid
Product Marketing Engineer, NXP

Development boards are an integral part of developing and testing for embedded systems. As more devices in industrial facilities, the home, and buildings become connected to the Internet of Things (IoT), the need for streamlined and cost-efficient ways to design and innovate new technology has skyrocketed.

It is no secret that spending time testing connections on a breadboard, or worse, soldering, could cause massive headaches and undue stress when a developer's time could be better spent coding. Electronics development has evolved significantly, from the complexities of single-board computers to the user-friendly convenience of development boards. Development boards have integrated essential components like microcontrollers, power supplies, and pre-programmed functionalities onto a single, compact platform. This shift has democratized electronics development, making it faster, more affordable, and easier for a wider range of creators to learn. To further boost developer efficiency, NXP's FRDM development boards [1] as shown in **Figure 1** come with an MCX MCU, a standard form factor and headers, easy access to the MCU's I/Os, and an on-board debugger, which makes it ready to use.

NXP's MCX portfolio, which includes the MCX A series, is part of the MCUXpresso Developer Experience and has many ready-to-use tools for engineers, such as its MCUXpresso IDE. NXP's enhanced FRDM boards allow for faster prototyping while also offering flexibility and scalability, not just from MCX A microcontrollers but across the entire MCX portfolio.

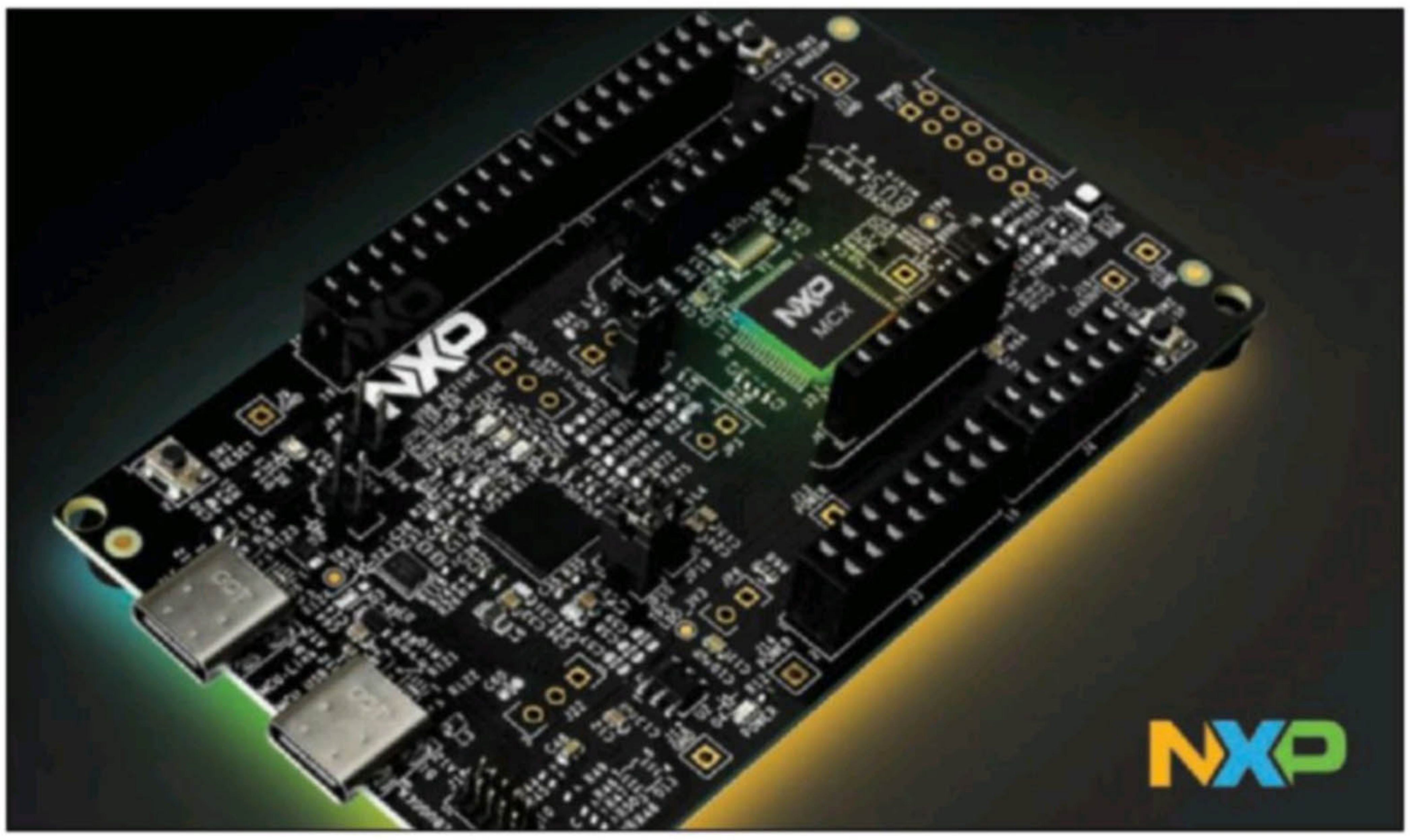


FIGURE 1
NXP's FRDM Development boards feature MCX MCUs. (Image source: NXP)

DEVELOPER CHALLENGES

Worldwide competition and adaptation are key factors in creating today's development board products. Because an increasing number of devices are currently entering the market, they are all fighting for the same customers, and several challenges have arisen as a result. For instance, the supply chain has impacted the availability of products that engineers need for development. As a result, these materials are in high demand and are becoming obsolete. Due to all these uncontrollable factors and the need to stay current in the market, all developers need product migrations and fast development.

This factor makes tools and enabling software an essential part of rapid development and is one area where NXP offers customers added value to the products for both MCUs and MPUs. NXP offers more than just semiconductors; offerings span solutions, proof of concept designs, an integrated developer experience, and highly flexible evaluation boards that enable the industry to keep products coming out of their factories. Additionally, NXP has options to make its MCU portfolio accessible to different developer profiles as shown in **Figure 2**, from students who are building short-term projects to major corporations that have a wide range of complex requirements and much longer development cycles.

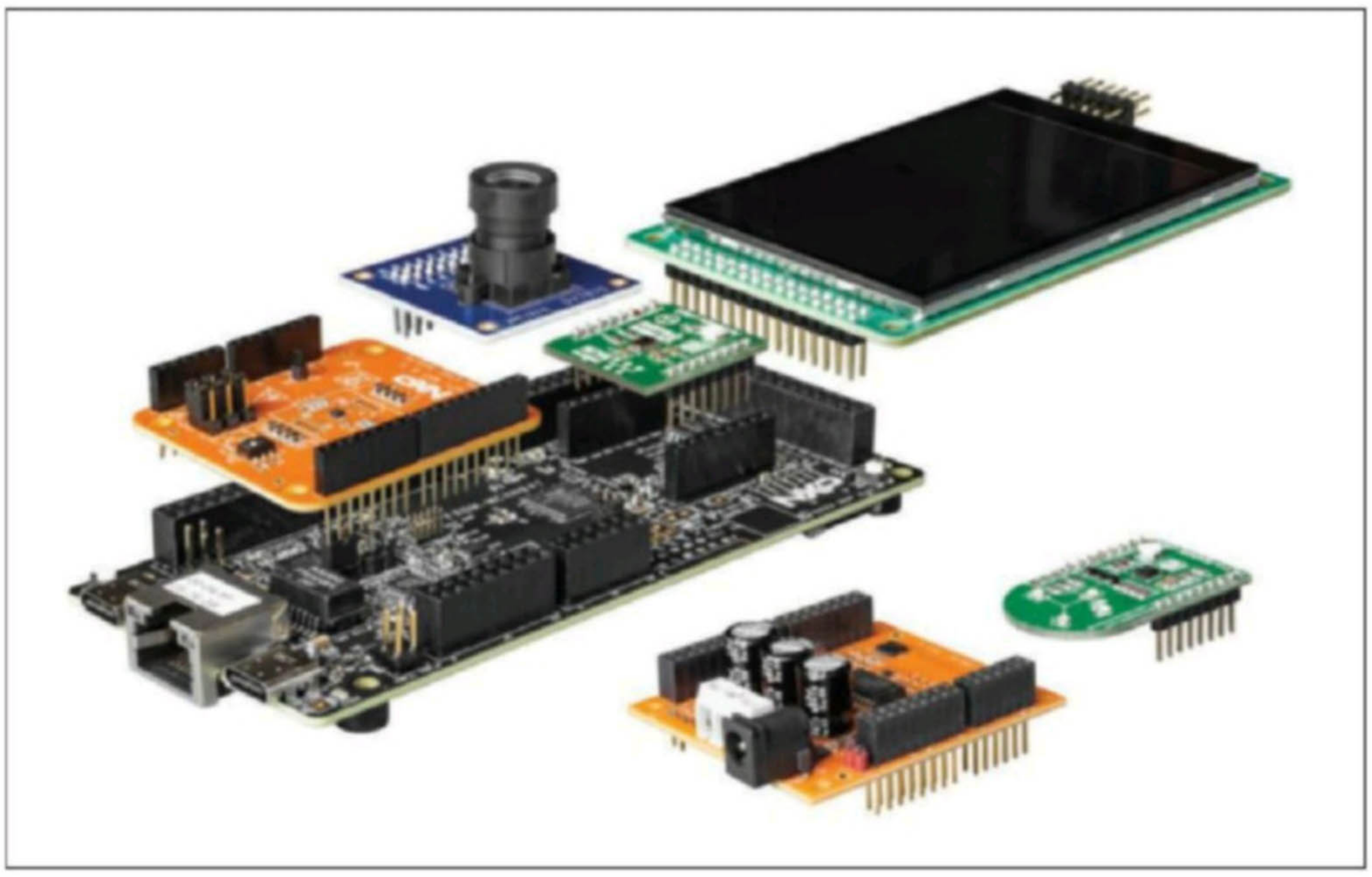


FIGURE 2
NXP FRDM development boards offer expansion options to meet the requirements of many different profiles. (Image source: NXP)

NXP FRDM PLATFORM

With the release of the MCX A14x and MCX A15x in the MCX A [2] series shown in the **Figure 3**, it is important to have a development board to streamline innovation with them. The FRDM platform gives developers flexibility and the capability of rapid prototyping. The FRDM boards come standard with an on-board MCU-link debug probe to make debugging code effortless and industry-standard headers that will allow for compatibility with existing devices without the headache of finding special cables or adapters. The ability to plug and play with external components like sensors, motor drives, and more allows developers to skip the hassle of worrying about compatibility with common devices.

EMPOWERING CREATIVITY

Leading hardware is only half of the battle that engineers face when creating new technology. They must also tackle intricate software and algorithm development, manage ever-growing complexity while making devices more user-friendly, address ethical considerations, and navigate cost and time constraints. Regulatory compliance adds other obstacles, demanding adherence to various safety standards and frameworks.

The MCUXpresso Developer Experience offers tools to jump-start development so that developers can focus on other challenges. This allows developers to spend more time writing software specific to their needs instead of creating their own SDKs and APIs. NXP offers both an Eclipse-based IDE and simple integration with Visual Studio Code via a plugin.

Additionally, the Application Code Hub contains snippets of code developed by NXP's in-house engineers and experts. Here, developers can see a range of simple and complex uses for the FRDM boards, which removes many of the obstacles that come with starting a new project. The available software packs and demo applications are also easy to import into IDEs.

Continuous integration and deployment are commonplace in modern software design. The MCUXpresso SDK [3] comes with pre-written CMake scripts that can be integrated with an engineer's preferred development, build, and deployment tools. The entire MCX portfolio can use this tooling, and it gives engineers the freedom to develop and innovate in a method that fits their preferences. The Application Code Hub that NXP offers also gives examples, code snippets, and software kits developed by NXP's in-house engineers.

The MCUXpresso Developer Experience with the FRDM development platform includes a multitude of expansion boards available on the Expansion Board Hub with drivers compatible with the MCUXpresso SDK. It also includes examples of how to integrate with them and functional and robust debugging tools. The FreeMASTER Run-Time Debugging Tool allows for debugging in real time, monitoring variables while a system is running without intruding, and visualization of data in multiple forms.

ALL-PURPOSE MCU POWER

So, why are developers increasingly turning to all-purpose MCUs for enhanced efficiency in their projects? All-purpose MCUs are versatile chips that offer a multitude of benefits for various applications. Affordable and versatile, they can handle complex projects with diverse functions even on a tight budget. They simplify development by reducing component integration and management, leading to faster time-to-market.

NXP's new MCX A series, including the MCX A14x and MCX A15x, are excellent all-purpose MCUs for any engineer. They are low-cost, have a minimal footprint, and offer autonomous, low-power peripherals, which help to make the newest innovations accessible to every engineer.

NXP
MCX A Series

FIGURE 3
NXP developer boards featureing the MCX A14X and MCX A15X microcontrollers, give developers flexibility for rapid prototyping. (Image source: NXP)

Both the MCX A14x and MCX A15x feature an Arm Cortex-M33 core, with the MCX A14x running at clock speeds up to 48MHz and the MCX A15x running at clock speeds up to 96MHz. The MCX A series comes in many different memory amounts with up to 1MB of flash. Both feature BLDC and PMSM controls and a number of integrated sensor interfaces (MIPI-I3C, I^2C, SPI), as shown in the block diagram in **Figure 4**.

One of the MCX A series' defining features is the selection of autonomous peripherals that allow for the offloading of tasks from the microcontroller so it can run at a lower frequency and save power. These peripherals include DAC, mixed signals ADC, serial comms with built-in buffers, a programmable data collection range, and FlexPWM.

ENERGY EFFICIENCY

A key capability of the MCX A14x and MCX A15x is that they can operate efficiently with low power. RAM can be retained down to very low-power modes. This allows the MCU to sit idle when not being used and consume very little power without losing data stored in RAM. It also can remain in communication with a sensor or external device and begin running again when a certain signal is received.

Both of the new MCX A series have a low-power cache controller attached to the Arm Cortex-M33 core. Like any CPU cache, it can store data and instructions in a place much closer to the microprocessor and with lower latency than direct memory access. More importantly, when it's not being used, it can be used as instruction memory.

Low power consumption is an important feature in industries like manufacturing and agriculture, as environmental regulations govern emissions and power usage. An embedded system for monitoring temperature or humidity can run on very little power, with the signal triggering a function based on the measurement.

DEFINING REQUIREMENTS

Developers are receiving a growing list of requirements that they didn't use to have to stay ahead of competitor devices appearing in the market. Additionally, developers also have the demand to create and implement more value-added features for differentiation. The current trend is toward devices becoming smarter and more secure, which is driven by customer concerns and their desire for more functionality. However, crucially, these developments are also mandated by new cybersecurity legislation. Developers need to adapt quickly to all these new requirements, often in areas where they may have limited experience.

FUTURE OF EMBEDDED DEVELOPMENT

As the landscape of embedded development keeps evolving, all-purpose MCUs and development boards empower engineers to translate their ideas into reality with greater efficiency and reduced complexity. These versatile devices, equipped with advanced features and functionalities, allow developers and engineers to create with no restrictions. NXP's MCX portfolio [4] is developed with cutting-edge technology and is at the forefront of power-efficient devices for use in all forms of engineering. Explore more of NXP's MCU offerings [5] to see how you can create with the future in mind and develop technology across a variety of industrial and IoT markets.

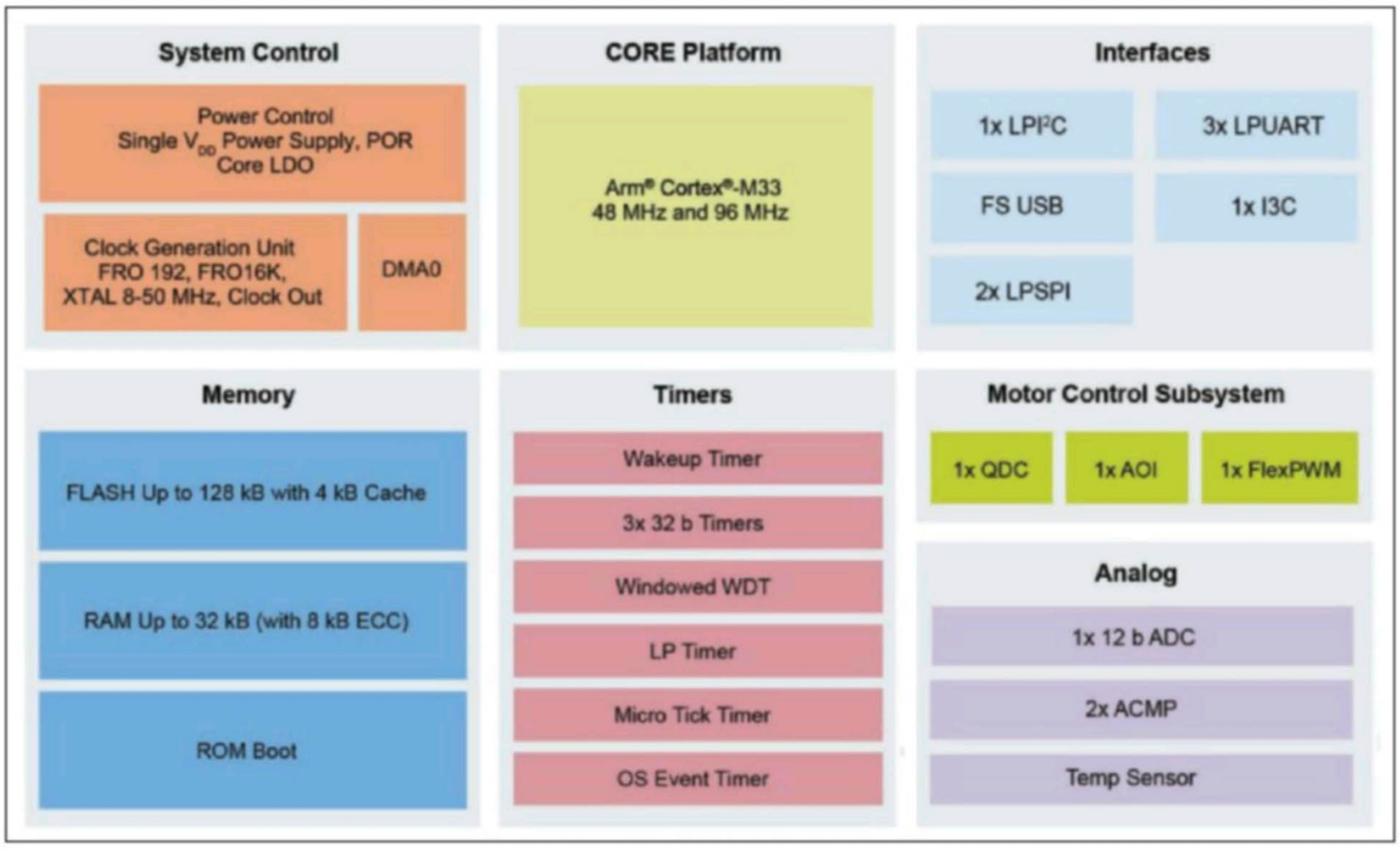

System Control
Power Control
Single V_DD Power Supply, POR
Core LDO
Clock Generation Unit
FRO 192, FRO16K,
XTAL 8-50 MHz, Clock Out
DMA0
CORE Platform
Arm® Cortex®-M33
48 MHz and 96 MHz
Interfaces
1x LPI²C
3x LPUART
FS USB
1x I3C
2x LPSPI
Memory
FLASH Up to 128 kB with 4 kB Cache
RAM Up to 32 kB (with 8 kB ECC)
ROM Boot
Timers
Wakeup Timer
3x 32 b Timers
Windowed WDT
LP Timer
Micro Tick Timer
OS Event Timer
Motor Control Subsystem
1x QDC
1x AOI
1x FlexPWM
Analog
1x 12 b ADC
2x ACMP
Temp Sensor

FIGURE 4
The MCX A series comes in many different memory amounts with up to 1MB of flash. The series also features BLDC and PMSM controls and a number of integrated sensor interfaces (MIPI-I3C, I^2C, SPI), as shown in the block diagram (Image source: NXP)

Additional materials from the author are available at:
www.circuitcellar.com/article-materials
References [1] and [5] as marked in the article can be found there.

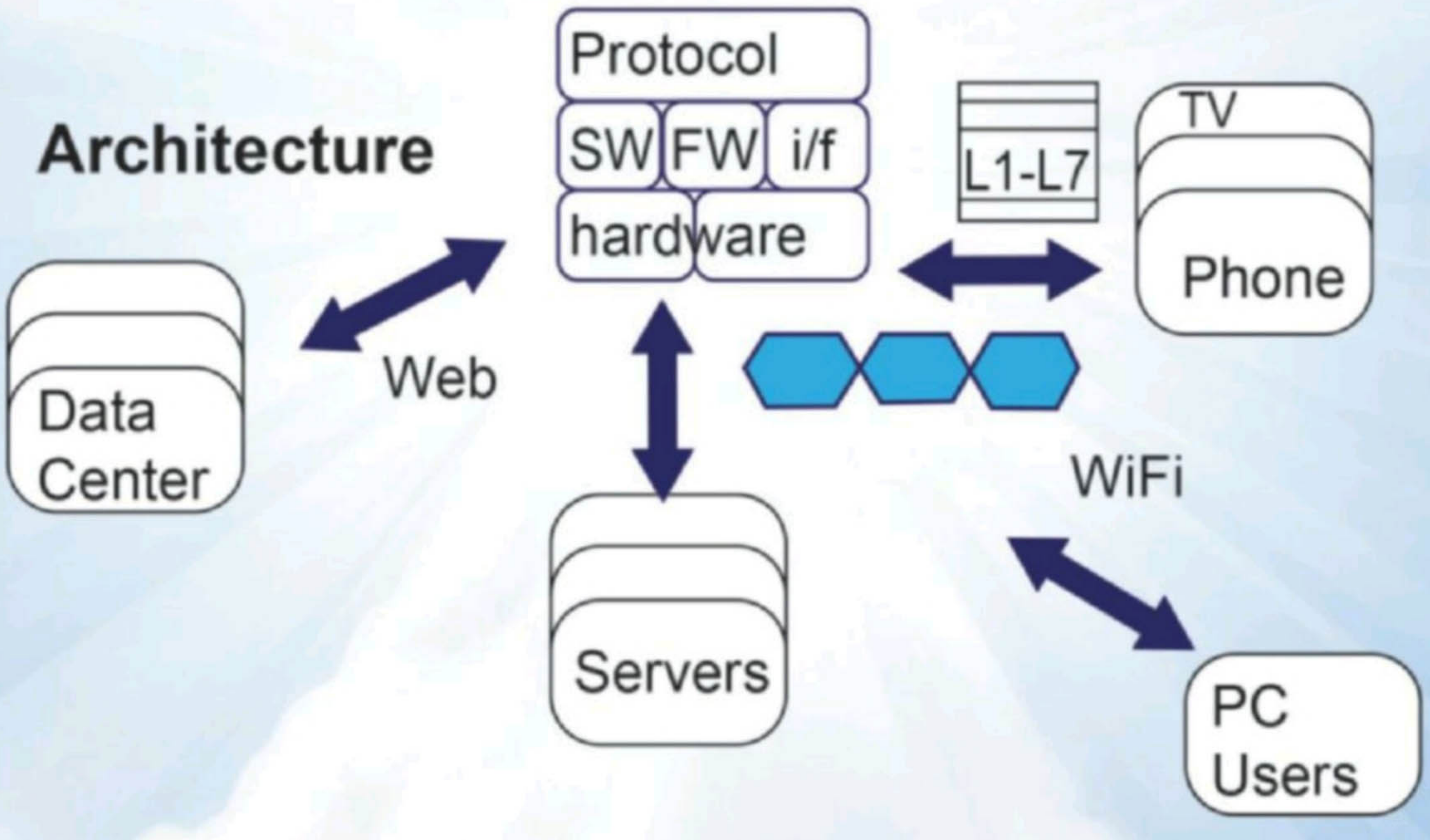
Resources
NXP | www.nxp.com

ABOUT THE AUTHOR

Mónica Cid is a product marketing engineer in NXP's Secure Connected Edge business. She has been with NXP since 2022, starting as a technical marketing engineer for edge processors. Prior to this, she worked as a software developer in the manufacturing industry. She has a bachelor's degree in artificial intelligence engineering and a master's degree in project administration. Mónica lives in Guadalajara, Mexico, and is passionate about technology and projects that help make life easier and better for everyone.

SeaFire® Micros, Inc.

IP & Technology for Cloud Networks


Architecture
Protocol
SW
FW
i/f
hardware
L1-L7
TV
Phone
Web
Data Center
WiFi
Servers
PC Users

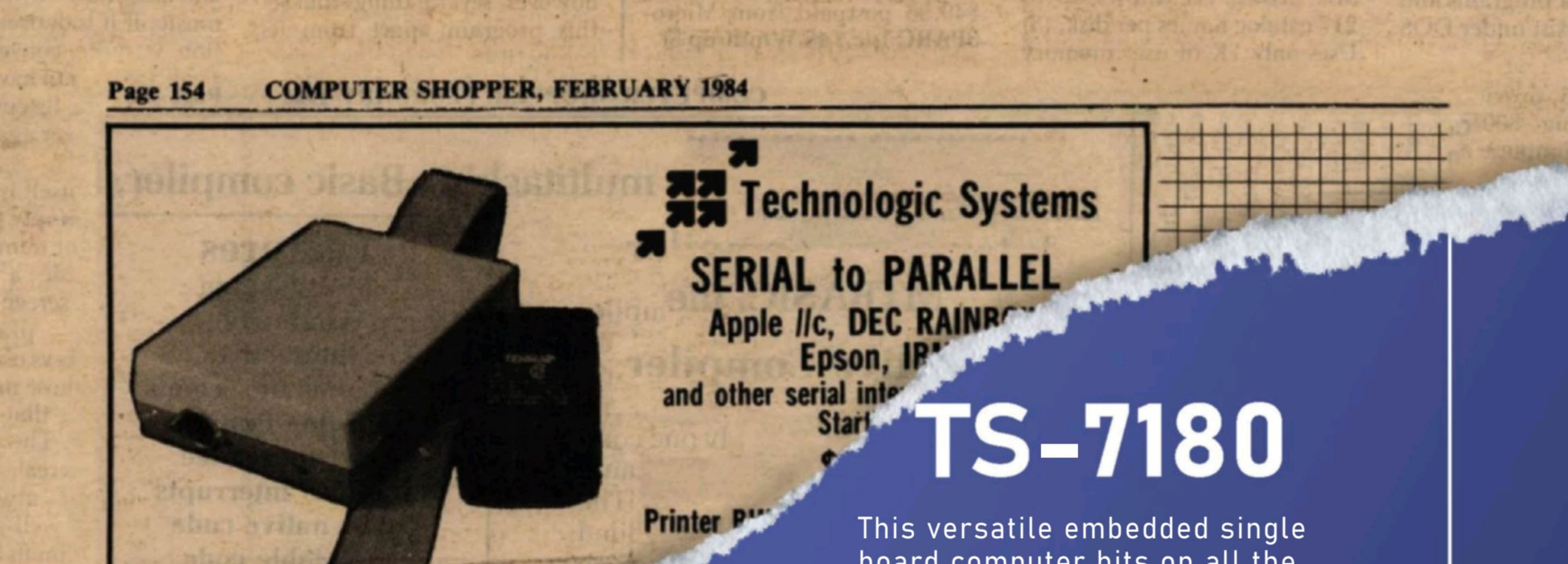
Design through Production
Military — Commercial

Hardware (Analog/Digital)
Firmware • Software • FPGA

Patented Network Technology | Unique Cloud Technology
No Compete Government Procurement
No Compete Government Funding

20% DISCOUNT on 1st Purchase

info@seafire.com | seafire.com



Page 154 COMPUTER SHOPPER, FEBRUARY 1984

Technologic Systems

SERIAL to PARALLEL

Apple //c, DEC RAINB

Epson, IB

and other serial inte

Start

Printer P

TS-7180

This versatile embedded single board computer hits on all the main points for a low power, cost effective, Internet-of-Things (IoT) capable, and ready-to-deploy OEM board with an emphasis on data integrity.

Rev.D
TS-7180

NXP i.MX6UL SOC with a 696 MHz Arm® Cortex®-A7 core

512 MB DDR3 RAM (1 GB DDR3 RAM Option Available)

4 GB MLC eMMC Flash, configurable as pSLC for added reliability

Connected with WiFi, Bluetooth, MultiTech and XBee/NimbeLink

Easily Track Assets with GPS Radio and Cellular Modem

Celebrating 40 Years

From humble beginnings in a garage in Ohio, EmbeddedTS has grown to become a leader in rugged embedded computing solutions. Since 1984, we've proudly remained family-owned and dedicated to delivering high-quality, reliable products.

Every board we sell is still designed, shipped, and supported from right here in the United States, ensuring you get top-tier craftsmanship.

Made in the USA
with global parts

embeddedTS
embeddedTS.com